Hindawi
Security and Communication Networks
Volume 2021, Article ID 2897565, 14 pages
https://doi.org/10.1155/2021/2897565

Research Article

SESCon: Secure Ethereum Smart Contracts by Vulnerable Patterns' Detection

Amir Ali,[1] **Zain Ul Abideen**,[2] **and Kalim Ullah**[3]

[1]*School of Cyber Science and Engineering, Xian Jiaotong University Xian, Shaanxi 710049, China*
[2]*Department of Computer Science, Xian Jiaotong University Xian, Shaanxi 710049, China*
[3]*Department of Computer Science, CECOS University of IT and Emerging Sciences, Peshawar, Pakistan*

Correspondence should be addressed to Amir Ali; amir.ali@stu.xjtu.edu.cn

Received 11 July 2021; Revised 12 August 2021; Accepted 20 August 2021; Published 21 September 2021

Academic Editor: Farhan Ullah

Ethereum smart contracts have been gaining popularity toward the automation of so many domains, i.e., FinTech, IoT, and supply chain, which are based on blockchain technology. The most critical domain, e.g., FinTech, has been targeted by so many successful attacks due to its financial worth of billions of dollars. In all attacks, the vulnerability in the source code of smart contracts is being exploited and causes the steal of millions of dollars. To find the vulnerability in the source code of smart contracts written in Solidity language, a state-of-the-art work provides a lot of solutions based on dynamic or static analysis. However, these tools have shown a lot of false positives/negatives against the smart contracts having complex logic. Furthermore, the output of these tools is not reported in a standard way with their actual vulnerability names as per standards defined by the Ethereum community. To solve these problems, we have introduced a static analysis tool, SESCon (secure Ethereum smart contract), applying the taint analysis techniques with XPath queries. Our tool outperforms other analyzers and detected up to 90% of the known vulnerability patterns. SESCon also reports the detected vulnerabilities with their titles, descriptions, and remediations as per defined standards by the Ethereum community. SESCon will serve as a foundation for the standardization of vulnerability detection.

1. Introduction

E-commerce has now prevailed on the Internet to purchase everything using digital cash, where a trusted third party can process and verify the transactions made on the Internet across the globe. However, the security concerns in online transactions are prevailing everywhere due to their centralized nature. For example, users can spend the same digital cash multiple times, hence causing the double-spend attack. Similarly, tampering of digital records through hacking or even by insiders has no 100% secured solution, yet. Therefore, to overcome this problem, blockchain was introduced in the bitcoin application [1] and provides a strong solution for the double-spending amount and immutability without any trusted party.

The main strong aspect of blockchain is to provide irreversible transparent transaction history on the distributed ledger. Anyone can see the complete history of user transactions. No one can change or alter the contents of this distributed ledger (blockchain). After a few years from the inception of bitcoin blockchain, researchers had realized the strong potentials of its Distributed Ledger Technology (DLT), which can be applied to other domains in parallel to its cryptocurrency applications. Vitalik Buterin provided a concept of smart contract (SC) to extend the blockchain applicability to other domains and introduced the Ethereum blockchain [2] and also introduced their own cryptocurrency, i.e., Ether. Smart contract has Turing complete language, i.e., Solidity, where any business logic can be defined and programmed. A smart contract is basically a piece of code that is deployed and stored permanently on blockchain with a unique address and executed automatically upon receiving some transactions.

There are two types of accounts in Ethereum blockchain, namely, Externally Own Account (EOA) and Smart Contract Account (SCA). An EOA is normally controlled through a private key that Ethereum users save in a secured place (Wallet). Anyone can create this account without paying any cost. This account can initiate different types of transactions in the Ethereum blockchain network, i.e., transferring/receiving Ethers (Ethereum cryptocurrency) from one account to another and signing transactions by its private key to prove its authorization. On the other hand, SCA is a little bit different in its nature. For example, to create this account we need some fractions of Ether and it has some associated code (written in Solidity/other languages), which will be deployed and stored on blockchain network permanently. We cannot remove/delete such an account from the blockchain network except to disable it. SCA can only initiate transactions when it receives transactions and do some action upon fulfilling the predefined criteria, hence providing some sort of automation. Both types of accounts have balance in the form of Ethers.

In the Ethereum network, millions of smart contracts have been deployed with a balance of billions of US dollars, which shows the potentials and trust of a lot of investors and business ventures. However, due to the irreversibility nature of its distributed ledger, we cannot change/alter SC's code if there would be some logical errors. If any vulnerabilities are found in SC after deploying it on blockchain, then it is very hard to fix its bugs by simply patching its source code like in conventional applications. Therefore, if there is any vulnerability found in SC, then we have to fix it and deploy it on the blockchain with a new address and also deactivate the previously deployed one.

SC holds billions of dollars and has been mostly utilized in the financial sector and related activities, hence attracting the attacker to steal its money. Therefore, it demands a strong security guarantee to prevent its breach. Unfortunately, it is very hard to produce bugs-free SC, since the developers are not trained and do not have so much experience in the blockchain environment. As a result, a large number of the critical vulnerabilities had been found in most famous smart contracts, i.e., the DAO [3], Frozen Ethers [4], and King of Ether [5], and caused loss of millions of dollars.

To secure the SC and detect its security vulnerabilities before deployment to blockchain, many researchers have contributed to the static and dynamic analysis of SC. Some has provided fuzzing solution (ContractFuzzer [6]), symbolic execution (Oyente [7], ZUES [8], and Securify [9]), fuzzing + symbolic execution (ILF [10]), taint analysis (Slither [11], NPChecker [12]), XPath of vulnerabilities patterns (SmartCheck [13]), etc. The main problems with these static analysis solutions are the high percentage (up to 70%) of false positives and false negatives. Furthermore, reporting of their detection is based on their own defined taxonomy of patterns, which may not be compatible with Ethereum defined standards patterns. Therefore, it is very difficult to evaluate such tools and measure their true positives/negatives against standard defined patterns.

Well-known vulnerabilities patterns in smart contracts have been defined by experts from the Ethereum community [14]. They [14] give each vulnerability with its title, description, relationship with CWE (Common Weakness Enumeration), remediation, samples of vulnerable contracts, and their fixed versions. If we consider these patterns and obey their guidelines, we can develop and deploy smart contracts with confidence and provide bug-free smart contracts. Our research work is based on these standards, and we have followed their guidelines during the detection of vulnerabilities. However, most of the state-of-the-art tools mentioned above could not detect such well-known critical vulnerabilities in smart contracts, or/and their output is not reported in a standard way [14]. They just detect the existence of some keywords with parameter values and give alarms (false positives/negatives) without checking them intelligently. For example, the most well-known attack of DAO on a smart contract even in its simple form [14] could not be detected by the SmartCheck [13] and/or other tools.

In this paper, we are going to provide an intelligent solution toward a static analysis of smart contracts. Our solution is based on taint analysis [15] and XPath of vulnerabilities patterns technique [13]. We first take the source code of the smart contract and convert it to an XML parse tree in the form of Abstract Syntax Tree (AST) where each node represents different constructs (statements, expressions, loops, conditions, variables, etc.). From AST, we become able to detect all its variables, statements, and vulnerable keywords/patterns through XPath queries. Then we pass it to our taint analysis module, where its dependent and nondependent variables, local and state variables, and their flows are detected. We then make vulnerable patterns as defined in standard vulnerability patterns [14]. As we follow the guidelines provided by Ethereum experts [14], therefore, our tool SESCon is aware of all vulnerabilities patterns and thus provides very high accurate vulnerability detection. Our main contributions in this research work are as follows:

(i) We have reduced false positives during vulnerabilities detections by providing an intelligent tool, i.e., SESCon, based on taint analysis which statistically analyzes the smart contract's source code written in Solidity language.

(ii) Since the state-of-the-art tools have defined their own taxonomy of vulnerabilities and detected source code against them, which could not be compared with the standards defined by the Ethereum community, and it became hard to evaluate their efficiency, we are the first, as per our knowledge, to detect vulnerabilities against these standards [14], hence providing a foundation for standardization toward detection and reporting vulnerabilities.

(iii) We have also applied our tool to detect vulnerabilities in real contracts. Our results show that a large number of vulnerabilities still exist in almost all contracts (i.e., 99%).

The rest of the paper is organized as follows: the related work is presented in Section 2; Section 3 gives some background to understand the domain of our research work. The detailed solution of our architecture is described in Section 4; in Section 5, experimental results, evaluation, limitation, and implementation are illustrated; and we finally concluded our research work with future extension in Section 6.

2. Related Work

In this section, we have reviewed some state-of-the-art works, which are related to static analysis directly or indirectly.

ContractFuzzer [6] has detected some vulnerabilities by instrumenting the EVM and defined some test oracles against such vulnerabilities. It has generated a lot of fuzzing inputs from contract ABI specs and invoked the contracts and analyzed their execution log. Oynte [7], on the other hand, by using symbolic execution techniques with Z3 constraint solver on contract bytecode, has detected three types of bugs, TOD, Timestamp, mishandle exception, and some level of reentrancy, whereas ZUES [8] used abstract interpretation and symbolic execution techniques to provide a formal solution for the correctness and fairness of SC in XACML style. It performed static analysis on IR to determine verification predicate points for assertion checking. Similarly, Securify [9] uses the symbolic execution, and also its algorithm has learned the semantics of code and then detected vulnerabilities patterns against violation/compliance patterns. In the first attempt by ILF [10] to fuzz the contract after symbolic execution, it has used the imitation learning framework by using a neural network to model its fuzzing policy to generate effective inputs for the sequence of transactions. However, ILF needs constructor valid parameters to deploy which causes hindrance toward automatically fuzzing the contract under test. Slither [11] converts Solidity contract to Intermediate Representation (SlithIR) using SSA (Static Single Assignment) form and applied data flow and taint tracking techniques to detect vulnerabilities. However, we cannot use Slither on a large dataset of contracts if they have a different version of the Solidity compiler.

NPChecker [12] provides the solution for some payment call bugs by employing the nondeterminism behavior of Ethereum (using bytecode to LLVM IR), without any already know pattern-based vulnerabilities, but has more false positives which need some human interaction to verify. It has adopted taint analysis for local and global variables of SC to explore information flow. However, it has not provided the solution for common arithmetic issues (i.e., integer overflow). Meanwhile, other works [16–22] also addressed the DoS, reentrancy, and Transaction Origin vulnerabilities on EVM bytecode patterns by using the symbolic execution and machine learning techniques without considering the standard patterns of vulnerabilities. EtherTrust [23] provided a static solution through formally defining security properties (reachability) of the semantic of SC (bytecode) into Horn clauses and queries were solved by the Z3 SMT solver. They focused on single reentrancy (SE) and independent miner transaction (MI) bug-related vulnerabilities. VerX [24] introduced a new symbolic execution engine in which delayed abstraction is applied to verify the functional safety properties of SC. They first formalized the temporal safety properties of SC by extending Solidity language (i.e., always and once). Then SC is instrumented to achieve the reachability property and, at the end, the symbolic execution engine has performed its function. Vandal [25] worked on EVM bytecode to abstract interpretation using declarative language, i.e., Souffle, and then generate logic relation. They used a logic-driven approach for defining security vulnerabilities. However, defining the new vulnerabilities needs expertise, which is not user-friendly. There were also some other works [26, 27], which are presented to provide semantic and formal correctness of smart contract at bytecode and smart contract functional level.

To summarize and conclude the related works, most tools (ContractFuzzer, Oynte, ZUES, ILF, Securify, Vandal, etc.) that have used symbolic execution or fuzzing or formally verified the security of smart contracts are operated on bytecode and/or source code. However, their solutions have a lot of overheads during the conversion of source code to some Intermediate Representations (IR), generating test cases for fuzzing, or dynamic running for symbolic executions to detect vulnerabilities. These solutions have not achieved 100% code coverage and hence left some well-known vulnerabilities undetected. Although Slither [11] used taint techniques to construct their Intermediate Representations constructs (SlitherIR), their solution is not directly applied to the source code of smart contract. Also, their solution is time-consuming due to the construction of its IR and SSA (Static Single Assignment) form. NPChecker [12] also uses taint techniques, but it is only related to payments activities and its payments bugs, whereas our SESCon tool is used to find all the security vulnerabilities as defined by the Ethereum community [14]. The most related to our work is SmartCheck [13], which uses static analysis through AST of Solidity source code by translating its grammar in ANTLR and detecting patterns of vulnerabilities by XPath technology. However, it just detects some keyword or some simple pattern; however, its false positive rate is very high, i.e., 69%. Further, for complex patterns, SmartCheck fails. We have extended SmartCheck with taint analysis so that we can detect vulnerabilities patterns intelligently. Another most related work [28] also uses XPath techniques to detect vulnerabilities in smart contracts, but their work is limited for integer overflow.

3. Background

Blockchain can be described as an append-only distributed database (called ledger) where data (transactions) are stored in chronological order. A group of transactions on this ledger are packed into a block and each block is cryptographically linked to the previous block, hence making a chain of blocks (called blockchain). The most important aspect of this technology is that its architecture is decentralized, which means there is no single server that would be

responsible to process and manage/store all transactions [1]. Therefore, blockchain technology is inherently secured from DoS/DDoS, since there is no central point of failure. Anyone can join this peer-to-peer network just by installing its open-source software and participate without any permission, which is called permissionless blockchain, i.e., bitcoin [1] and Ethereum [2]. There are also some permissioned blockchains, where access control layers have been introduced to provide additional security in the system [29–33] so that certain actions must be performed by only authorized participants.

In general, when a user wants to interact with the blockchain, he/she just sends a transaction (i.e., Alice sends 1 bitcoin/Ether to Bob), and this transaction is passed to a pool of transactions (mining pool), where special nodes (miners) select transactions randomly as per their priority (mostly have high fees) and put them into a logical block and produce a hash of the whole block with some restriction of its contents (leading number of zeros) with a puzzled algorithm. When a miner solves such a puzzle of leading zero to produce the hash of the block, he broadcasts this newly generated block to a P2P blockchain network (bitcoin, Ethereum, etc.). All the other nodes update their locally stored blockchain ledger, after the verification and validation of this new block. This puzzle is often called a consensus algorithm which is normally described as Proof of Work (PoW, in case of bitcoin blockchain) or Proof of Stack (PoS, in case of Ethereum 2.0 blockchain) [2]. All the transactions in any block are stored in Merkle Tree form [34], to achieve fast processing time and low storage space. A simple logical illustration of the blockchain is depicted in Figure 1.

However, to extend the blockchain functionality toward automation, the smart contract was provided [2]. Basically, a smart contract is a piece of program code written in any computer programming language (i.e., Solidity [35], Vyper [36]). This code is permanently deployed on the blockchain after compilation into bytecode and gets executed in Ethereum Virtual Machine (EVM) when certain conditions (written in the smart contract) are met. So, there is no third party to control its execution and hence provide a kind of automation (see details in Figure 2).

However, the main problem with smart contracts is that, when it is deployed on the blockchain, we cannot update it or patch it, if some vulnerabilities are found. However, there are some types of actions like Suicide and SelfDestruct which can disable your contract after transferring its balance to a newly deployed contract with the new address. Smart contract development is in its infancy stage, and that is why developers are not well aware of its complexities in terms of the blockchain environment, which leads to a lot of vulnerabilities.

4. SESCon Architecture

In this section, we have presented our SESCon (Secure Ethereum Contracts) solution architecture. First, we described its overview, then we have explained its modules, namely, Vulnerable Patterns module, XPath module, and the Taint module, and, at the end, we present our solution in algorithm form.

4.1. Overview. First of all, in the Vulnerable Patterns module, we have generated vulnerable patterns from the dataset of samples of smart contracts [14]. Details of this module are given in this section. In XPath module, we feed smart contract's source code to get its Intermediate Representation (IR) and then find some vulnerable patterns by utilizing the XPath queries. However, we do not totally depend on XPath, because it leads to false positives in the case of complex rules [13]. Therefore, we have introduced the Taint module on the output of the XPath module. In this module, we first identify its local and global variables, the most important being the state variables which can change the status of SC. We then captured the variables flow and its dependency graph [38] as per our defined algorithm. When we have captured all the information about source code, variable flow, and dependency graph, we then compare the SC under test with the vulnerability patterns defined by the Ethereum community [14]. Finally, SESCon also generates a report about the vulnerabilities patterns found in the source code of the Solidity file with its location, line number, and its potential solutions (please see Figure 3).

4.2. Vulnerable Patterns' Module. The purpose of this module is to generate vulnerable patterns of smart contracts in light of samples of vulnerable smart contracts as provided by the Ethereum community [14]. Since we have known vulnerable patterns of smart contracts in advance, therefore, we just use them to construct vulnerable patterns. This module consists of five tasks as shown in Figure 4.

4.2.1. Sample Contracts. Vulnerable smart contract samples are collected along with their fixed solutions [14]. These fixed solutions help the developer to correct the source code and make the smart contract fixed from vulnerability.

4.2.2. Preprocessing. The aforementioned vulnerable samples have gone through preprocessing steps, where we manually convert those samples into proper standard smart contract form so that we can fully capture its Intermediate Representation (IR).

4.2.3. Representation Learning. The IR consists of an Abstract Syntax Tree (AST), where we have identified the vulnerable keywords and labeled them. After the parsing of AST, we get the dependency graphs and their tokens to learn the potentials vulnerabilities [39].

4.2.4. Classification. In the classification phase, we have assigned the severity of each vulnerable pattern to high, medium, or low. For example, the Outdated Compiler Version vulnerability is considered as low, whereas the reentrancy vulnerability is considered high and so on.

4.2.5. Vulnerable Patterns. In the end, all patterns have been stored in our database, which has been used as a standard

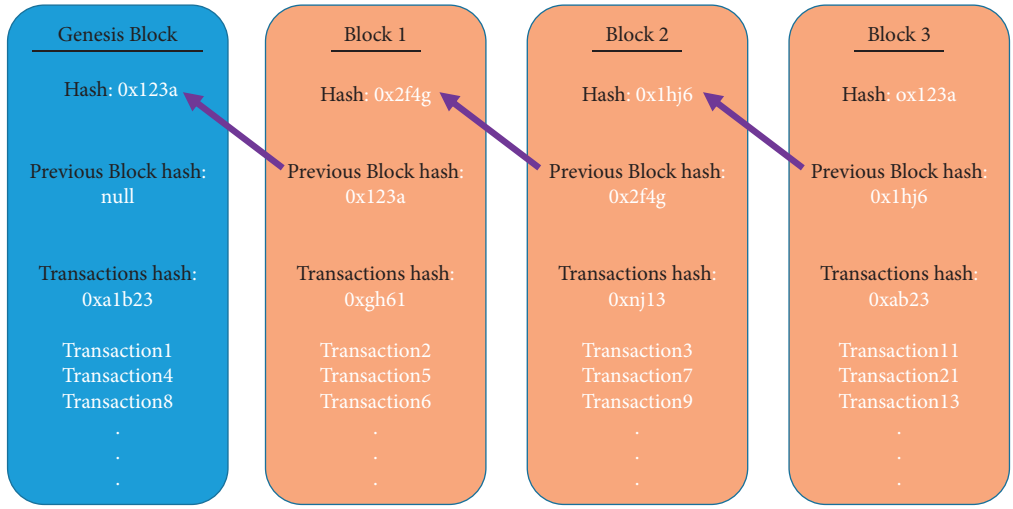

Figure 1: Overview of the blockchain [1].

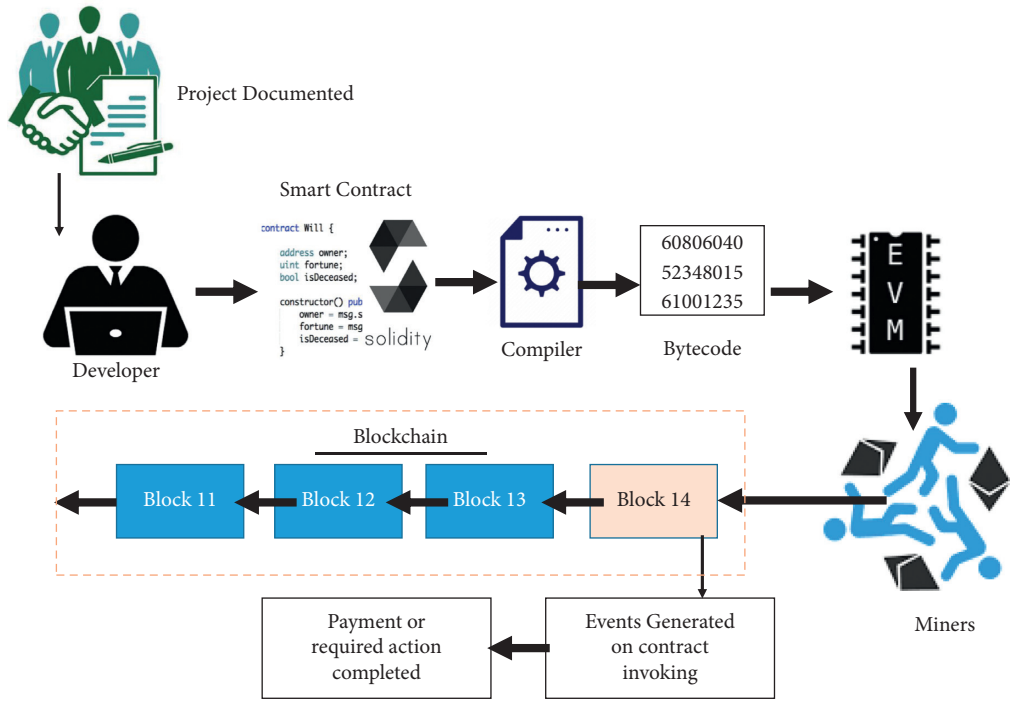

Figure 2: Smart contract execution cycle [37].

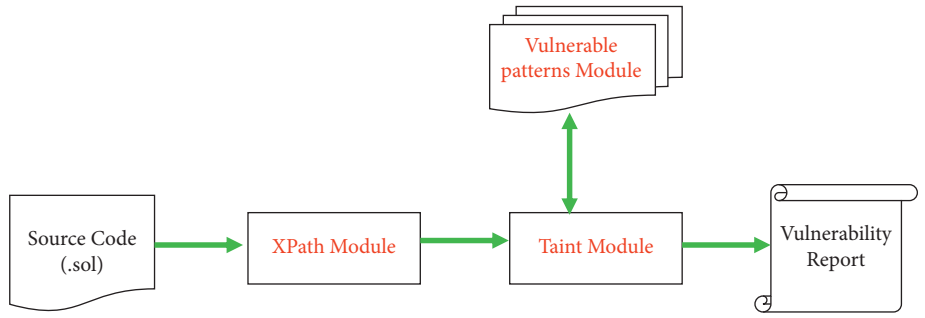

Figure 3: Overview of the SESCon architecture.

Figure 4: Vulnerable patterns' module.

reference to detect vulnerability in real-world smart contracts.

4.3. XPath Module. In this module, first, we take the source code of SC written in a Solidity language (.sol file) and convert it to its XML parse tree in the form of Abstract Syntax Tree (AST). After getting the AST of the source code, now we have all its keywords, i.e., statements, expressions, and variables. Using XPath queries, we have detected vulnerabilities as in [13]. This module takes the Solidity (.sol) source code file as input. We first achieved its AST as shown in Figure 5.

Then using Solidity grammar and ANTLR [40], we have constructed the XML parse tree. This tree was then analyzed through XPath [41] to detect some vulnerable keywords and patterns. By parsing source code as an XML tree, we have achieved 100% code coverage and all elements are matched through XPath queries. It also provides the vulnerabilities locations and line numbers as XML attributes. Therefore, we can easily indicate the vulnerabilities in our source code of smart contract under test. XPath module and the following taint module are given in Figure 6.

4.4. Taint Analysis Module. This module takes the output of the XPath module as input and extracts Control Flow Graph, Protected Functions, local variables and state variables, and Data Dependency Graph. Here we see which variables can/cannot read/write and especially cause a change in the status of smart contract by modifying the state variables. This module provided us with all the internal and external flow of contracts, i.e., which function can change the state variables, and which functions are dependent. We have provided here a sample contract with all the information which can be visualized as given in Figure 7.

Figure 7 shows that the smart contract is comprised of the main contract (apexSolid.sol), especially its purchase function, which is interacting with functions and other contracts, i.e., SafeMath and ERC721. The data flow from one function to another function is shown by an arrow. The green arrow is for the internal flow of data, whereas the orange arrow indicates the external flow of data. We have also extracted the protected, private, and public methods and the variables that can change the value of Smart Contract's state variable and which method is payable. In the end, we make patterns and label them as SmartContractPatterns, and these patterns will go through test. The same procedure is also applied to sample standard vulnerable contracts to extract their patterns. Finally, we compare these two patterns and show vulnerabilities, if comparing results was true. Most of the tools could not detect even the known critical

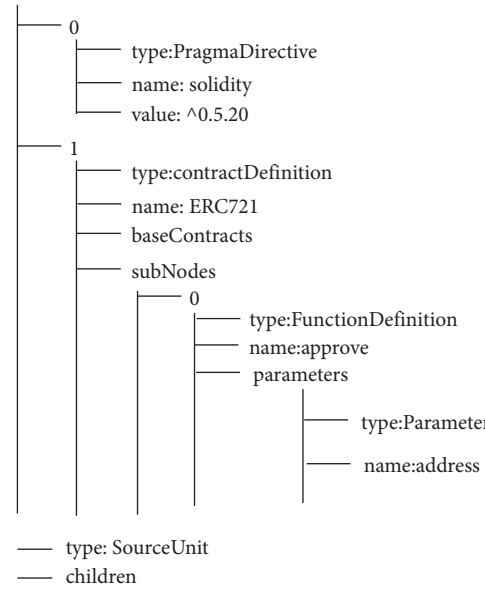

Figure 5: Abstract syntax tree (AST).

vulnerabilities in a smart contract. They just detect the existence of keywords with some parameter values and give their alarms (false positives/negatives) without checking intelligently. For example, the most well-known attack of DAO on a smart contract [14], even in its simple form (given in Figure 8), could not be detected by [13].

First, we give sample standard DAO code (Figure 8) to different tools and check whether they detect DAO vulnerability or not. If smart contracts have this kind of vulnerability, most state-of-the-art tools have detected this vulnerability. However, when we feed the corrected fixed version of the DAO contract (Figure 9), suggested by the Ethereum community [14], to tools, they almost failed and gave us false positives about the reentrancy attack. To provide a solution for these kinds of issues, there is a need for some intelligent algorithms, which must be aware of data flow before and after the execution of each statement. For example, if we feed a vulnerable standard sample pattern of a contract to the tool, it must report it as vulnerable. However, when we feed the fixed version (Figure 9) of the vulnerable contract (after removing vulnerable code from the contract), then the tool must show this new fixed contract (Figure 9) as a secure contract. For this purpose, we have introduced an intelligent algorithm and implemented it in the form of the SESCon tool.

Therefore, our algorithm is not dependent only on just keywords and simple patterns. These patterns are a well-known sequence of instructions that lead to some vulnerabilities in smart contract, for example, calling external

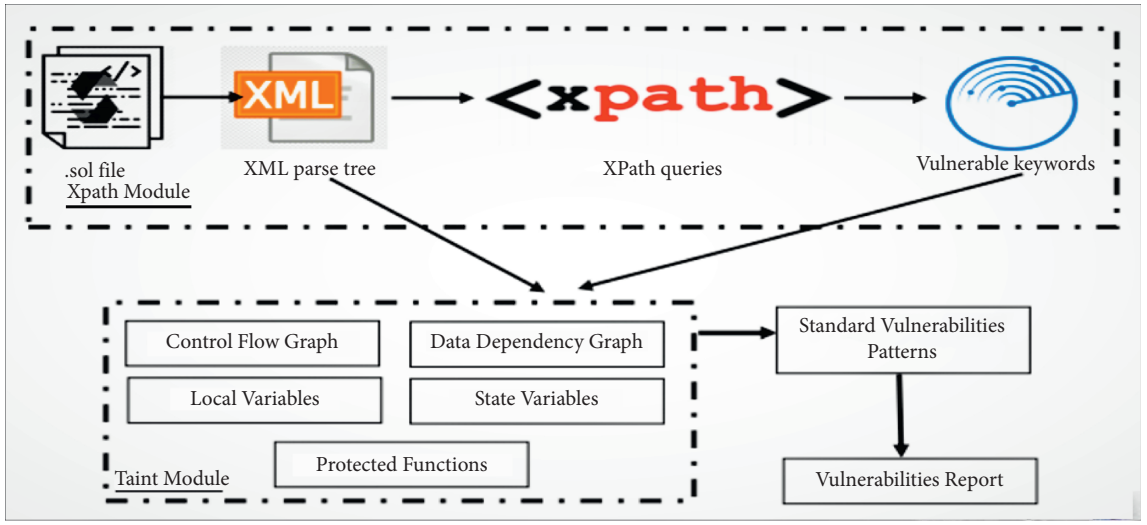

Figure 6: Detailed components of SECon.

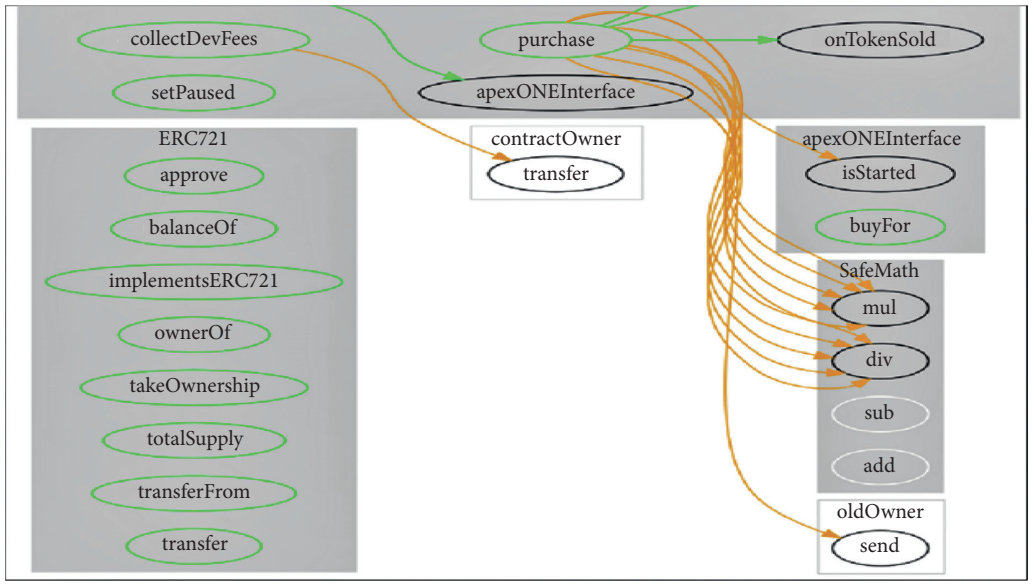

Figure 7: Information flow between different functions of the smart contract.

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

Figure 8: Simple DAO withdraw function.

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

Figure 9: Simple fixed DAO withdraw function.

contracts, insecure usage of SelfDestruct instruction, and checking the return value of message call. We have provided a context-aware solution to detect the vulnerable smart contract variables. Hence, we checked and compared the vulnerable keywords and patterns from its Control Flow Graph and Data Dependency Graph and see that whether it is exploiting the best practices of smart contract or these keywords/patterns are the actual requirement of business logic of state variables. This gives us accurate true positives about detected vulnerabilities as per standards defined by the Ethereum community.

In Algorithm 1, first, we give the .sol file (smart contract) to the SESCon tool, which extracts its Abstract Syntax Tree (AST) and then converts it to the XML parse tree. From lines 5 to 10, required parameters/variables and functions are extracted which may or may not affect the state of a smart

contract. In line 11, the standard sample patterns of vulnerable contracts are collected from the local repository. Then (at line 12) each pattern is compared with the target contract under test. If the pattern is found (line 13), it is added in detectedVulList (line 14) and its occurrence is stored in the source code of the contract (line 15). At the end (line 18), a standard report is generated which shows the title of vulnerability, its description, and its solution to correct.

With the help of this algorithm, we can check not only the vulnerable keywords but also the ability to detect vulnerable patterns, intelligently. Our algorithm is aware of all contract flows graphs, i.e., money flow graph (using payable functions), dependency graph of state variables, and local variables along with their tainted values. This makes our algorithm more accurate than the state-of-the-art tools.

5. Experimental Results

In this section, we have presented our experimental results. First, we present some statistics of our dataset of contracts, then we show the results that we have achieved on testing samples provided by the Ethereum community for vulnerability detection. In the end, we have shown our results against real contracts.

5.1. Statistics of the Dataset of Smart Contracts. Dataset of our contracts under static analysis can be categorized as standard samples defined by the Ethereum community [14] and real contracts. However, here we only focused on real contracts to describe their statistics. So, these real contracts can be viewed as balance of contract and line of source code. Regarding balance, we have observed that most of the contracts have zero balance. The remaining nonzero contracts and their balances (Ethers) distribution can be seen in Figure 10. Only one contract has more than one million Ethers. In the case of lines of source codes, smart contracts range from 20 lines to 3500 lines. We have also seen that most (70%) of the contracts have compiler version 4.x, 25% contracts are compiled with 5.x version, and a very few number of contract (5%) were with compiler version of 6.x (see Figure 11). For some tools (Slither, Securify, etc.), if the contract has version 0.4.x and we have installed 0.5x compiler, then their tool fails at an initial stage. Therefore, they are dependent on the compiler version. However, our tool is not dependent on the compiler version; it just takes the source code of the smart contract and then starts analyzing for vulnerabilities.

5.2. Testing against Standard Vulnerability Samples. Vulnerabilities patterns in smart contracts have been defined [14] by experts from the Ethereum community. We have tested the state-of-the-art tools (SmartCheck, Solhint, Securify, Slither) on these provided standard samples and compared our results with these state-of-the-art tools. The summary is presented in Table 1, where "yes" means vulnerabilities are detected accurately, "no" means they are not detected, and "partially" means they are detected with some false positive. Overall SESCon has shown outstanding performance among the state-of-the-art tools, and almost 90% of defined vulnerability patterns are detected, while other tools could not reach more than 55% (see Figure 12).

To evaluate the performance of our proposed approach toward detection of vulnerabilities in sample standard contracts, we have used three evaluation metrics, namely, precision, recall, and $F1$-score, which are normally used in pattern recognition [43]. So in our case, they are defined as follows: precision is the proportion of sample standard smart contracts that are correctly detected as vulnerable among all the sample vulnerable smart contracts.

$$\text{Precision} = \frac{\text{smart contracts correctly detected as vulnerable}}{\text{all smart contracts detected as vulnerable}}. \tag{1}$$

Recall, in our case, is the proportion of sample standard smart contracts that are correctly detected as vulnerable among all the sample really vulnerable smart contracts.

$$\text{Recall} = \frac{\text{smart contracts correctly detected as vulnerable}}{\text{all really vulnerable smart contracts}}. \tag{2}$$

There is another method with which we can test our accuracy of results, that is, the $F1$-Score (F-score or F measure). This is a harmonic means of our precision results and recall results and measures by the following formula:

$$F1 - \text{score} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}. \tag{3}$$

Here, we summarized our evaluation of the accuracy of results against standard vulnerabilities (Table 2).

The reason behind the same values for precision, recall, and F1-Score against each tool is that, in our dataset, we have only standard sample smart contracts that are already defined as "really" vulnerable by the Ethereum community. In this dataset, there is no smart contract that is not "really" vulnerable. To achieve realistic results, we have tested our tool against real contracts and we measured the accuracy against real contracts. Then, we obtained the understandable results as presented in Table 3.

5.3. Testing against Real Contracts. We have also tested our tool against more than 8000 real contracts which we have downloaded from Etherscan [44] using JSoup API [45]. Most of the downloaded contracts (85%) have a balance of zero, and the remaining contracts have a balance of more than 1 Ethers and up to 1 million Ethers. SESCon has analyzed 8125 contracts, and most of the contracts have shown vulnerabilities, which are shown in Figure 13.

The main reason behind the exceptional results is that we have designed our solution in light of standard vulnerability patterns [14] and followed their suggested solutions to generate standard vulnerable patterns. This will help us to detect the vulnerability patterns in real-world smart contracts with great accuracy, while in other solutions, they have devised their own patterns and their tools detect them accordingly without considering the standard vulnerable patterns.

(1) Read the smart contract
(2) Extract the abstract syntax tree
(3) Convert AST to XML path using XPath queries
(4) Store locations of each statement which L_1, L_2, ... L_n
(5) Get control flow graphs (cf1, cf2)
(6) Get dependency graph, dg1, dg2
(7) Get local variable (lv1, lv2)
(8) Get state variable (sv1, sv2, \enleadertwodots svn)
(9) Get payable function (pf1, pf2)
(10) Get nonpayable function (npf1, npf2, ...\enleadertwodots npfn)
(11) Load standard patterns of vulnerabilities p_1, p_2, ..., p_n
(12) **for** each (pi) compare dgi in given smart contract **do**
(13) **if** foundPattern **then**
(14) detectVulList.add (pi)
(15) locationsList.add (Li)
(16) **end if**
(17) **end for**
(18) Generate report

Algorithm 1: Detecting vulnerabilities in the smart contract using SESCon.

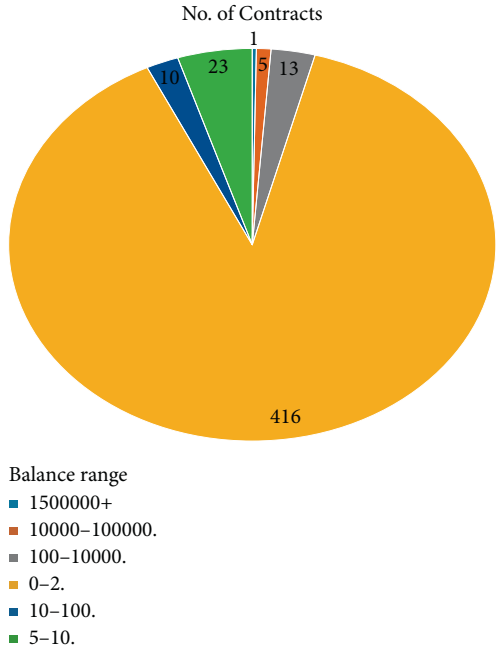

Figure 10: Balance in Ethers distribution in contracts.

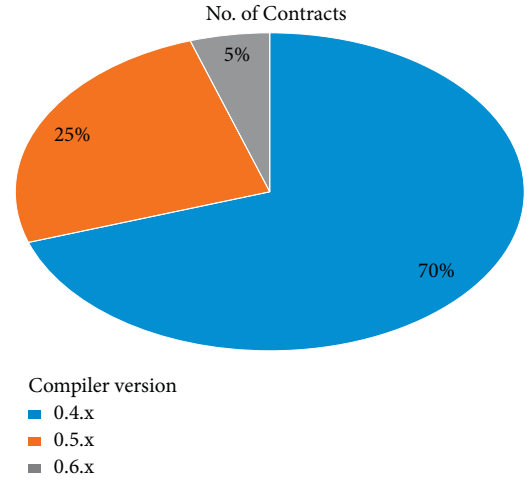

Figure 11: Compiler versions' distribution.

Another reason is that some vulnerability needs an in-depth analysis of smart contract code and its context. This means that only vulnerable patterns or keywords detections are not enough to report vulnerabilities, but we have to consider where the vulnerable code is present. This is because, sometimes, a vulnerable pattern is not causing vulnerability, if the said pattern existed in source code along with some security precautions; hence, it should not be reported.

5.4. Evaluation and Limitation. To validate our tool, we give a sample vulnerable smart contract from standard patterns [14] and check whether the SESCon can detect it or not, and our tool gave 100% accurate true positive detection. Then we give a fixed version of the vulnerable smart contract to SESCon, which shows 100% accurate true negative detection. After that, we have tested our tool on real smart contracts. During testing our tool against real contracts, it is revealed that still a lot of contracts have well-known vulnerabilities, namely, DAO, Transaction Order Dependency (TOD), trx.origin, and block.timestamp, and uses some assembly code. Among them, DAO type vulnerabilities were at the top and tx.original was the second most discovered vulnerability.

The main limitation with our tool is that it performs static analysis against standard vulnerabilities patterns defined by the Ethereum community; therefore, we cannot detect zero-day exploits. Due to the unavailability of samples of contracts for certain standard vulnerabilities patterns, we have also some false positives, which we are trying to overcome in our future work. Since our tool is

Table 1: Comparison of vulnerability detection.

Vul. ID [14]	Solhint [42]	Securify [9]	Slither [11]	SmartCheck [13]	SESCon
SWC-100	Yes	Yes	Yes	Yes	Yes
SWC-101	No	Yes	Yes	No	Yes
SWC-102	Yes	No	Yes	No	Yes
SWC-103	Partially	Yes	Yes	Yes	Yes
SWC-104	Yes	Yes	Yes	Yes	Yes
SWC-105	Yes	Yes	Yes	No	Yes
SWC-106	Yes	Yes	Yes	No	Yes
SWC-107	Partially	Yes	Yes	Partially	Yes
SWC-108	Yes	Yes	Partially	Yes	Yes
SWC-109	No	Yes	No	No	Yes
SWC-110	No	No	Yes	No	Yes
SWC-111	Yes	No	Yes	Yes	Yes
SWC-112	Partially	Yes	Yes	Partially	Yes
SWC-113	Yes	Partially	No	Yes	Yes
SWC-114	Partially	Yes	Yes	Partially	Yes
SWC-115	No	Yes	No	Yes	Yes
SWC-116	Yes	Yes	Partially	No	Yes
SWC-117	No	No	No	No	Yes
SWC-118	No	No	Yes	No	Yes
SWC-119	No	Partially	Yes	No	Yes
SWC-120	Yes	No	No	No	Yes
SWC-121	No	Partially	Yes	No	Yes
SWC-122	No	No	Yes	No	Yes
SWC-123	No	No	Yes	No	Yes
SWC-124	No	Yes	No	Yes	Yes
SWC-125	No	Partially	No	No	Yes
SWC-126	No	Partially	Yes	No	Yes
SWC-127	Partially	No	Yes	Partially	Yes
SWC-128	No	No	Partially	Partially	Yes
SWC-129	Partially	Partially	No	No	Yes
SWC-130	No	Yes	Partially	No	Yes
SWC-131	Partially	No	Yes	No	Yes
SWC-132	Partially	Yes	Yes	No	Yes
SWC-133	No	No	Partially	No	Partially
SWC-134	Partially	No	Partially	No	Partially
SWC-135	No	Partially	No	No	Partially
SWC-136	No	No	No	No	Partially

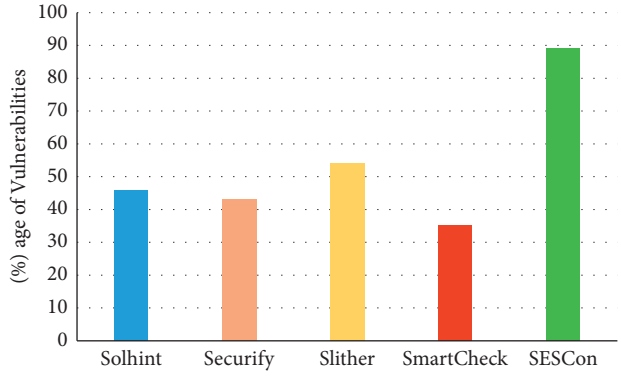

Figure 12: Testing against standard vulnerability samples.

dependent on the source code of the smart contract, therefore, it cannot be valid and applied to the bytecode of the smart contract. There are also some works [46, 47], which claimed that their tools can be used to evaluate the existing static analyzer tools for Ethereum smart contracts. We have tried to incorporate our tool into their work and evaluate our tool, but due to some technical errors during building their work, we failed to integrate

Table 2: Accuracy results against standard vulnerabilities.

Models	Precision	Recall	*F*1-score
Solhint [42]	0.5263	0.5263	0.5263
Securify [9]	0.6956	0.6956	0.6956
SmartCheck [13]	0.6153	0.6153	0.6153
Slither [11]	0.7777	0.7777	0.7777
SESCon	0.8918	0.8918	0.8918

Table 3: Accuracy results against standard vulnerabilities.

Vul. ID	Precision	Recall	*F*1-score
SWC-100	0.880	0.961	0.919
SWC-101	0.880	0.961	0.919
SWC-102	0.880	0.961	0.919
SWC-103	0.880	0.961	0.919
SWC-104	0.880	0.961	0.919
SWC-105	0.880	0.961	0.919
SWC-106	0.880	0.961	0.919
SWC-107	0.880	0.961	0.919
SWC-108	0.880	0.961	0.919
SWC-109	0.880	0.961	0.919
SWC-110	0.880	0.961	0.919
SWC-111	0.880	0.961	0.919
SWC-112	0.880	0.961	0.919
SWC-113	0.880	0.961	0.919
SWC-114	0.880	0.961	0.919
SWC-115	0.880	0.961	0.919
SWC-116	0.880	0.961	0.919
SWC-117	0.880	0.961	0.919
SWC-118	0.880	0.961	0.919
SWC-119	0.880	0.961	0.919
SWC-120	0.880	0.961	0.919
SWC-121	0.880	0.961	0.919
SWC-122	0.880	0.961	0.919
SWC-123	0.880	0.961	0.919
SWC-124	0.880	0.961	0.919
SWC-125	0.880	0.961	0.919
SWC-126	0.880	0.961	0.919
SWC-127	0.880	0.961	0.919
SWC-128	0.880	0.961	0.919
SWC-129	0.880	0.961	0.919
SWC-130	0.880	0.961	0.919
SWC-131	0.880	0.961	0.919
SWC-132	0.880	0.961	0.919
SWC-133	0.880	0.961	0.919
SWC-134	0.880	0.961	0.919
SWC-135	0.880	0.961	0.919
SWC-136	0.880	0.961	0.919

them. Further, their evaluation criteria are different from ours: we have made patterns to detect vulnerability while they just injected bugs and then detect them. We will consider them [46, 47] to investigate and evaluate our tool in the future.

5.5. Implementation. We have implemented SESCon in Java 8 and for lexical and syntactical analysis we have used ANTLR 4.8 and Solidity grammar 5.6 on Windows 10 64-bit platform on the core of i7 with RAM 4 GB. To find simple vulnerabilities patterns and keywords, we have used XPath 2.0 queries.

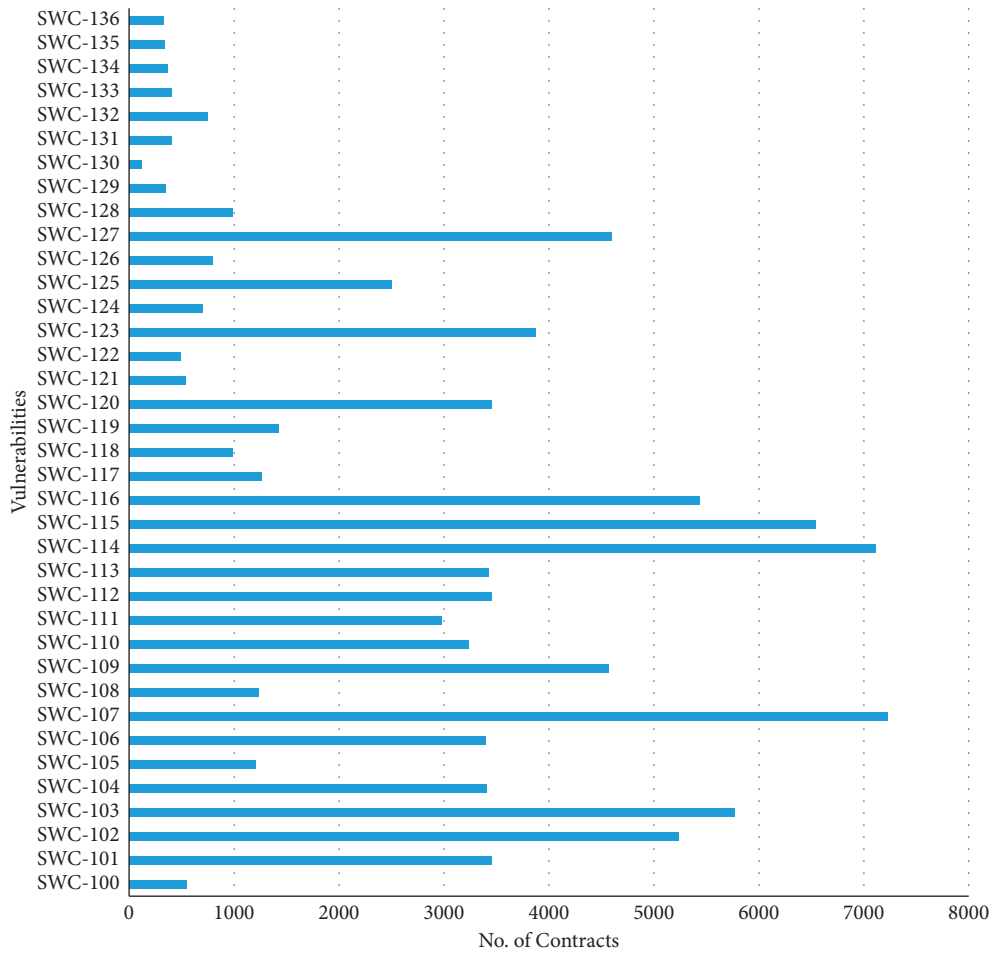

Figure 13: Testing against real contracts.

6. Conclusion and Future Work

In this paper, we have presented a solution to detect smart contract vulnerability through static analysis. Our solution is based on XPath and taint analysis. Most of the static analysis tools for smart contract give a large number of false positives. We have reduced such alarms with a hybrid approach of a combination of XPath and taint analysis. First, we convert a .sol file to its equivalent AST XML parse tree and apply the XPath query to find some simple vulnerabilities patterns. After that, we go through deep analysis by taint analysis techniques, where we have extracted state variable, local variable, their control flow, graph dependency of function, and payable and nonpayable functions to make some vulnerable patterns. Then we compare smart contract under test against the standard vulnerabilities patterns defined by the Ethereum community. Our tool outperforms other analyzers and can detect up to 90% of the known vulnerability patterns, accurately. We have also analyzed more than 8000 real contracts through our tool SESCon and found that a large number of vulnerable smart contracts still existed, which could be corrected by our tool. Our solution will provide a foundation toward the standardization in comparing and evaluating tools with standards vulnerabilities patterns.

Our work can be extended in many directions. The natural extension of our work is to reduce some false positives against the last four vulnerabilities (SWC-133 to 136) patterns. As we have performed static analysis on the Solidity source file, our next work would be to detect vulnerabilities analysis from the bytecode of smart contract. We are also planning how we could detect some zero-day exploits by some machine learning approaches. In the end, after consolidating the tool, we will provide our solution as an open source after some enhancements, so that early-stage researchers can be facilitated.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

[1] S. Nakamoto, "Bitcoin: a peer-to-peer electronic cash system," *Decentralized Business Review*, 2008, https://bitcoin.org/bitcoin.pdf.

[2] V. Buterin, "A next-generation smart contract and decentralized application platform," 2014.

[3] The Dao –2016: https://etherscan.io/address/%200xbb9bc244d798123fde783fcc1c72d3bb8c189413.

[4] Parity attack: https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html>.

[5] The King of Ether - https://www.kingoftheether.com/postmortem.html>.

[6] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, New York, NY, US, September 2018.

[7] L. Luu, D. H Chu, and H. Olickel, P. Saxena and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, Vienna, Austria, October 2016.

[8] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: analyzing safety of smart contracts," in *Proceedings of the Network and Distributed System Security Symposium NDSS*, San Digeo, CA, USA, February 2018.

[9] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Bunzli, and M. Vechev, "Securify: practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, Toronto, Canada, October 2018.

[10] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, London, UK, November 2019.

[11] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, May 2019.

[12] S. Wang, C. Zhang, and Z. Su, "Detecting nondeterministic payment bugs in Ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1–29, 2019.

[13] S. Tikhomirov, E. Voskresenskaya, and I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, Gothenburg, Sweden, May 2018.

[14] Vulnerabilities Patterns in Smart Contracts- https://swcregistry.io/docs/SWC-100.

[15] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," *NDSS*, vol. 5, pp. 3-4, 2005.

[16] Y. Chinen, N. Yanai, J. P. Cruz, and S. Okamura, "RA: Hunting for re-entrancy attacks in ethereum smart contracts via static analysis," in *Proceedings of the 2020 IEEE International Conference on Blockchain (Blockchain)*, IEEE, Rhodes Island, Greece, November 2020.

[17] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, and T. Peng, "Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts," in *Proceedings of the 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, September 2020.

[18] S. Schmeelk, B. Rosado, and P. E. Black, "Blockchain smart contracts static analysis for software assurance," *Lecture Notes in Networks and Systems*, vol. 284, pp. 881–890, 2021.

[19] N. He, R. Zhang, H. Wang et al., "EOSAFE: security analysis of EOSIO smart contracts," in *Proceedings of the 30th USENIX Security Symposium*, vol. 21, USENIX Security, Vancouver, Canada, August 2021.

[20] N. Ashizawa, N. Yanai, J. P Cruz, and S. Okamura, "Eth2Vec: learning contract-wide code representations for vulnerability detection on ethereum smart contracts," in *Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure*, Hong Kong, China, May 2021.

[21] N. F. Samreen and M. H. Alalfi, "Smartscan: an approach to detect denial of service vulnerability in ethereum smart contracts," 2021, https://arxiv.org/abs/2105.02852.

[22] K. L. Narayana and K. Sathiyamurthy, "Automation and smart materials in detecting smart contracts vulnerabilities in blockchain using deep learning," *Materials Today: Proceedings*, 2021.

[23] I. Grishchenko, M. Maffei, and C. Schneidewind, "EtherTrust: sound static analysis of ethereum bytecode," Technical Report D, Technische Universität Wien, Vienna, Austria, 2018.

[24] A. Permenev, D. Dimitrov, P. Tsankov, D. D. Cohen, and M. Vechev, "Verx: safety verification of smart contracts." in *Proceedings of the 2020 IEEE Symposium on Security and Privacy, SP*, San Francisco, CA, USA, May 2020.

[25] L. Brent, A. Jurisevic, M. Kong et al., "Vandal: a scalable security analysis framework for smart contracts," 2018, https://arxiv.org/abs/1809.03981>.

[26] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, "Ethor: practical and provably sound static analysis of ethereum smart contracts," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, November 2020.

[27] J. Schiffl, M. Grundmann, M. Leinweber, O. Stengele, S. Friebe, and B. Beckert, "Towards correct smart contracts: a case study on formal verification of access control," in *Proceedings of the 26th ACM Symposium on Access Control Models and Technologies*, New York, NY, USA, June 2021.

[28] E. Lai and W. Luo, "Static analysis of integer overflow of smart contracts in ethereum," in *Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy*, Nanjing China, January 2020.

[29] 2021 https://hyperledger-fabric.readthedocs.io/en/latest/whatis.html> Accessed on: 11.08.2021.

[30] 2021 https://docs.chainstack.com/blockchains/quorum> Accessed on: 11.08.2021.

[31] 2021 https://www.multichain.com/developers/permissions-consensus/> Accessed on: 11.08.2021.

[32] 2021 https://docs.corda.net/docs/corda-os/4.8.html Accessed on: 11.08.2021.

[33] J. Polge, J. Robert, and Y. Le Traon, "Permissioned blockchain frameworks in the industry: a comparison," *Ict Express*, vol. 7, no. 2, pp. 229–233, 2021.

[34] M. Szydlo, "Merkle tree traversal in log space and time," in *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques*, May 2004.

[35] Solidity -https://solidity.readthedocs.io/en/latest/>.

[36] Vyper -https://vyper.readthedocs.io/en/latest/>.

[37] S. Sayeed, H. Marco-Gisbert, and T. Caira, "Smart contract: attacks and protections," *IEEE Access*, vol. 8, pp. 24416–24427, 2020.

[38] A. Johnson, L. Waye, S. Moore, and S. Chong, "Exploring and enforcing security guarantees via program dependence graphs," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 291–302, 2015.

[39] X. Li, L. Wang, Y. Xin, Y. Yang, and Y. Chen, "Automated vulnerability detection in source code using minimum

intermediate representation learning," *Applied Sciences*, vol. 10, no. 5, p. 1692, 2020.
[40] 2020 Antlr–https://www.antlr.org/> Accessed on 20-06-2020.
[41] 2020 XPath-https://www.w3.org/TR/xpath20/> Accessed on 20-06-2020.
[42] 2020 Solhint: A solidity linting tool - https://github.com/protofire/solhint> Accessed on 20-06-2020.
[43] Precision and recall-https://machinelearningmastery.com/classification-accuracy-is-not-enough-more-performance-measures-you-can-use/>.
[44] 2020 EtherScan APIs-https://etherscan.io/apis> Accessed on 20-06-2020.
[45] 2020 JavaSoup Library-https://jsoup.org/%20Accessed on 20-06-2020.
[46] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, New york, NY, USA, July 2020.
[47] A. L. Vivar, A. L. Sandoval Orozco, and L. Javier García Villalba, "A security framework for Ethereum smart contracts," *Computer Communications*, vol. 172, pp. 119–129, 2021.