New Approach to Optimize Vulnerabilities Management of Smart Contract in Blockchain Network

Z.H. Randriamiarison[1], H. Razafimahatratra[2], N.R Razafindrakoto[3] and Y. Rhazali[4]

***Abstract*—Blockchain technology is gaining popularity today, replacing centralized data storage on a central server with a decentralized network of ledgers, thus ensuring secure information exchange. A smart contract is a program written on the blockchain that runs autonomously within the Ethereum virtual machine: a transparent and secure program, but once deployed on the blockchain network, it cannot be modified. Despite its advantage over other technologies, smart contract has become a prime target for hackers, making it difficult for developers to eliminate all vulnerabilities before its deployment. In this paper, we propose a static analysis approach aimed at reducing vulnerabilities in smart contract. Our method builds upon PASO (Parser for Solidity) and MSmart approaches: we developed a test lifecycle model for smart contracts and created a tool based on ANTLR4's G4 grammar. It involves both syntactic and lexical analysis to effectively detect bugs and vulnerabilities in smart contract. To validate our method, we used FDR (False Discovery Rate) and FNR (False Negative Rate) as evaluation metrics, data collected from SmartBugs and Etherscan. We validated our approach compared with MSmart and SmartCheck. We got higher FDR and improved FNR, indicating enhanced detection capabilities. After thorough analysis and extensive testing, our tool has proven to be both specific and high-performing.**

I. Introduction

In just a few years, the evolution and integration of Artificial Intelligence (AI) in the Information Technology (IT) have made significant strides. AI algorithm for detecting bugs in the blockchain network can help and to resolve many flaws and vulnerabilities [1]. Blockchain technology was introduced by researcher Satoshi Nakamoto in 2008 [2]. It is autonomous, independent, flexible and transparent [3]. Blockchain is defined as a decentralized network in which all data is stored in blocks [6]. Each block is linked to the previous one, forming a continuous chain known as the blockchain [4]. A block contains several components, including the block hash, Merkle root, target, transactions, smart contract, version, static data, and nonce [5]. Initially, the blockchain network was used solely for cryptocurrency transactions, particularly Bitcoin. Ethereum, a platform that supports blockchain-based contracts, was introduced by Vitalik Buterin in 2014 and launched in 2015 [6, 10]. Ethereum enables the execution of smart contract code. The concept of the smart contract was originally proposed by Nick Szabo in 1996 as a protocol for managing transactions between two parties [8]. Since 2015, blockchain technology has been applied in various sectors, including e-governance, gaming, and finance [9]. The most popular programming languages for creating smart contracts are solidity, vyper, and tiger [10].

The smart contract is automatically triggered when predefined conditions are met [4]. As of 2021, the estimated value circulating in the Ethereum blockchain network was around $130 billion [7]. Smart contracts are immutable, meaning they cannot be modified once deployed on the blockchain network [10]. However, the vulnerability of smart contracts presents significant risks [3]. A contract may contain bugs during execution [11], and hackers can exploit these flaws. Such vulnerabilities often arise due to several factors, including the developer's lack of experience, the relative novelty of the programming language, and the limited availability of tools for detecting vulnerabilities [10]. As a result, numerous vulnerabilities have been identified in the Ethereum network. According to [7], $60 million was lost in 2016, and this figure grew to $900 million by 2018 [3]. In 2020, many developers attempted to exploit existing flaws in smart contracts [11]. In 2021, approximately $1.55 billion was stolen from the network [10].

The goal of our work is to optimize vulnerabilities in a smart contract. Several approaches are available from different analyses such as dynamic analysis approach, static analysis and hybrid analysis approach [9]. In this article, we employ the statistical analysis approach. We aim to continue studying tools such as SmartCheck, MSmart, and PASO. Since the launch of Ethereum, the solidity programming language has undergone significant evolution. Earlier versions, such as solidity 0.6.*, were used to test the grammar rules of SmartCheck and MSmart. Currently, the language has advanced to version 0.8.*, which includes new features such as unchecked, reentrancy protection, abicoder, and try-catch blocks. The objective of this article is to update and clarify the grammatical rules to enable more accurate detection of vulnerabilities in the latest versions of solidity contracts. Features such as abicoder, timestamp, unchecked arithmetic, and reentrancy management (call, send, transfer) have been incorporated into our analysis. We used the ANTLR4 (Another Tool for Language Recognition, version 4) is a tool to translate grammar rules into tokens, lexers, and parsers. Additionally, we applied the FDR (False Discovery Rate) and FNR (False Negative Rate) metrics, as previously

[1]Zilga Heritiana Randriamiarison is with the Laboratory for Mathematical and Computer Applied to the Development Systems (LIMAD), University of Fianarantsoa, Fianarantsoa, Madagascar; (e-mail: zilgarandria@gmail.com).

[2]Hajarisena Razafimahatratra is with the Laboratory for Mathematical and Computer Applied to the Development Systems (LIMAD) University of Fianarantsoa Fianarantsoa, Madagascar; (e-mail: hajarisena@yahoo.fr).

[3]Nicolas Raft Razafindrakoto is a senior researcher at the Laboratory of Multidisciplinary Applied Research (LRAM), University of Antananarivo, Madagascar; (e-mail: rnraft@gmail.com).

[4]Yassine Rhazali is a senior researcher at the Laboratory RSILAB Ibn Tofail University, Kenitra Morocco; (e-mail: dr.yassine.rhazali@gmail.com)

used in MSmart, to evaluate the accuracy of our approach. The methodology, validation strategy, and results presented in this paper demonstrate the effectiveness and feasibility of our tool.

II. Description and Literature Review

A. Blockchain

Blockchain is a peer-to-peer autonomous network that does not require an intermediary to carry out transaction. Each transaction is executed on a decentralized ledger. Blockchain technology was introduced by Satoshi Nakamoto in 2008 [2]. Initially, it was used exclusively in the financial sector, most notably with Bitcoin. However, since 2015, with the introduction of smart contracts, blockchain has been applied to various fields such as e-governance, online voting, e-commerce, and finance [9].

Smart contract was first proposed in the 1990s by (Szabo, 1996) in [8], who described them as computerized protocols designed to execute transactions on a peer-to-peer network. A smart contract is automatically triggered when predefined conditions are met. It is typically developed using popular programming languages such as solidity, vyper, and tiger. Among these, solidity is the most widely used language for blockchain development [10]. Since the emergence of smart contracts, a large number have been deployed on blockchain networks. Each contract is executed by the EVM (Ethereum Virtual Machine) [1].

Certainly, blockchain technology offers many advantages over traditional Web2 technologies. However, one major limitation is that once a smart contract is deployed, it is immutable and cannot be altered. Solidity has undergone several version updates since its inception.

Vulnerabilities in a smart contract are a critical, as they can compromise the security of transactions within the blockchain network. Detecting errors in smart contracts is essential to ensure their reliability and safety. There are many types of possible vulnerabilities in a smart contract. In [13], the NCC (National Cybersecurity Center) states there are ten common vulnerabilities frequently found in smart contracts: Reentrancy, Control access, Integeroverflow, Unchecked Return values for Low Level Calls, Denial of Service (DoS), Bad Randomness, Front Running, Time Manipulation, Short adress, Calls to the Unknown Vulnerability. While many types of vulnerabilities may exist, these ten are the most well-known and commonly encountered in blockchain systems.

B. Literature Review

In [12], three types of approaches are identified for eliminating and studying vulnerabilities in smart contract code. The two approaches are the study of static vulnerabilities and the study of dynamic vulnerabilities. The third approach focuses on hybrid analysis. Among these, static analysis is the most widely used and considered the most efficient [12]. In this section, we will first examine the static analysis approach. Next, we will review the relevant literature.

Static code analysis involves the use of predefined rules to detect security issues without executing the code [14]. This method is very fast and secure [10]. It helps identify potential vulnerabilities in smart contracts, allowing developers to detect and fix issues during the development phase. Static analysis is considered both reliable and efficient, as smart contract can be tested without being deployed [4]. The process of creating a static analysis approach generally involves several steps, such as those described in [10]:

- Scanner code: this involves scanning the smart contract, analyzing the lexical, syntactic and semantic field;
- Flaw discovery: the tool will search for vulnerable codes in the smart contract;
- Bug classification: in this step we can identify the type of a bug or vulnerabilities in a contract.

Static analysis is one of the most commonly used approaches because it inspects the code without executing it [12]. This is one of the key advantages of static analysis, allowing it to offer more functionality compared to other analysis tools [10].

The approaches described in [6, 11, 15, 16] are based on static analysis. Each tool checks for vulnerable parts of the solidity language. They utilize ANTLR4 to generate an analysis tree. The researchers define error types, lexical fields, and the syntax of smart contract code within the grammar. Their approaches translate solidity source code into an XML-based intermediate representation, which is then verified using XPath models.

(Tikhomirov et al., 2018) validated their approach by comparing it with other tools to demonstrate its performance and effectiveness [6].

The approach of (Fei et al., 2023) aimed to improve the SmartCheck tool [11]. The researchers introduced new representation rules to enhance vulnerability detection and named their improved tool MSmart. They added a new class called Ttest, which manages folder localization for the input and output of smart contracts. The input contains solidity code, and the output is a folder containing files with .txt or .csv extensions. MSmart uses an XML parse tree to transform solidity code and can detect multiple types of vulnerabilities, including integer overflows, timestamp dependencies, self-destructs, delegatecall usage, and denial-of-service (DoS) vulnerabilities. In contrast, SmartCheck can only detect timestamp and self-destruct issues. The researchers compared the effectiveness of MSmart with that of SmartCheck.

(Pierro et al., 2020) developed a tool to test and analyze a smart contract before deploying it [16]. The researchers used ANTLR4 to traverse the AST (Abstract Syntax Tree) of a smart contract. Their approach involves inputting solidity code, after which the lexer module will give a token and decompress it into units. The parser module then generates the AST representation. Finally, the parser analyzes relevant elements to identify the components of the smart contract, such as functions, interfaces, events, and libraries. The researchers demonstrated the effectiveness, performance, and safety of their approach. The grammar code ends with the `.g4` file extension and can be used in various programming languages. The source code of the listener, lexer, and parser depends on the grammar used to generate them. Lexical and

syntactic rules for identifying errors in smart contracts are defined in the grammar file.

III. Proposed Approach Model

In this section, we propose a model for detecting bugs in solidity code using ANTLR4. ANTLR4 is a powerful parser generator used for reading, processing, and executing structured text. In our model, it is used to analyze solidity source code and generate an AST. We describe the steps taken to detect bugs in the code, from defining grammatical rules to evaluating test results. Fig. 1 shows the proposed model for detecting bugs in solidity code.

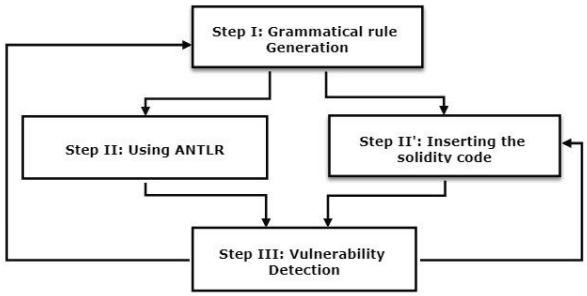

Figure 1. Approach model

The proposed model is based on four (4) steps:

Step I: Grammatical rule Generate

Step I consist of generalizing grammatical rules. We identify the lexical and syntactic structure of solidity code using a file with the `g4` extension. This file contains lexer rules for input and parser rules for generating the Abstract Syntax Tree.

Step II: Using ANTLR

In step II, ANTLR4 first takes the solidity grammar as input. It generates source code called the parser, which matches the sequence against the grammar and produces a parse tree as output [4].

Step II': Inserting the Solidity code

In step II', we insert the solidity source code from the dataset.

Step III: Vulnerability Detection

Step III involves vulnerability detection. First, the solidity source code is retrieved using the Lexer method. Then, the data is sent to the Token module. The parser module completes testing the code and displays the results. An error-free result from the solidity test indicates that the smart contract behaves as expected. Otherwise, the process returns to Step II. If the user is not satisfied with the vulnerability detection results, they can modify their own rule in Step I using the g4 extension.

IV. Validation of the Strategy

This section discusses the experimental setup, test scenarios and results obtained using our approach. The experiments aim to assess the accuracy, efficiency and user-friendliness of smart contracts, as well as the limits of our approach.

We developed a tool using the Python programming language under the Visual Studio Code IDE (Integrated Development Environment). The source code is available on the GitHub platform [17].

The experiments were conducted on an HP laptop equipped with an Intel Core i7-4610M processor, 12 GB of RAM, the Ubuntu 23.04 operating system, and the Ethereum Ganache Image node (a local blockchain network test environment).

A. Case Study

To evaluate the performance and accuracy of our approach, we created several scenarios, including reusing the Etherscan dataset previously utilized by MSmart. Our testing is divided into two projects. Project 1 consists of 20 contracts, 10 of which have timestamp dependency vulnerabilities, while the remaining 10 appear to be vulnerable. Project 2 also includes 20 smart contracts, 12 of which are labeled as vulnerable; however, in reality, none of these 12 contain vulnerabilities, and the other 8 contracts are truly vulnerable.

The test involves using the same dataset previously tested by MSmart. The goal is to assess our approach using the parameters established by MSmart. Within SmartBugs, there are various types of bugs, including Access Control, Arithmetic, Bad Randomness, Denial of Service, Front Running, Other, Reentrancy, Short Address, Time Manipulation, and Unchecked Low-Level Calls.

In a solidity grammar, each rule consists of a name, followed by a colon, then its definition, and ends with a semicolon. Fig. 2 shows a portion of our g4 extension grammar rule written in ANTLR4 syntax for Step I.

```
grammar Solidity;
sourceUnit
: (pragmaDirective | importDirective | contractDefinition)* EOF ;
pragmaDirective
: 'pragma' pragmaName pragmaValue ';' ;
pragmaName
: identifier ;
pragmaValue
: version | expression ;
version
: versionConstraint versionConstraint? ;
versionOperator
: '^' | '~' | '>=' | '>' | '<' | '<=' | '=' ;
versionConstraint
: versionOperator? VersionLiteral ;
importDeclaration
```

Figure 2. Extract from the code of the g4 grammar

We consulted the grammatical rules using the g4 extension from [6, 15, 16]. Table I illustrates a comparison between the MSmart, SmartCheck, PASO and our approaches.

TABLE I. Comparison Between Approaches

Criteria	Approaches			
	SmartCheck [6]	***PASO [16]***	***MSmart [11]***	***Our approach***
Version of Solidity tested with the approach	Solidity version <0.6.04	Solidity launched before September 2021	No tool available	Up to 0.8.*
Size of grammar file in Kb	31.1	11.2	Not Available	32
Grammar rules available	Yes	Yes	No	Yes

We have examined the grammatical rules used by SmartCheck [6] and PASO [16]. MSmart is an enhanced version of SmartCheck, featuring the addition of a Ttest class. However, we were unable to access the version created by (Fei et al., 2023) in [11]. We found that their tool versions were more recent. We have incorporated timestamps, reentrancy, abicoder and unchecked parts into our grammatical rule g4. A code excerpt is illustrated in Fig. 3.

```
sourceUnit
  //: ( pragmaDirective | importDirective | contractDefinition)* EOF ;
  : (timestamp | pragmaDirective | importDirective | contractDefinition)* EOF ;

timestamp
  : 'now' | 'block' | 'block' ('.' ('timestamp' | 'number'))? '+' INT unit;

unit: 'seconds' | 'minutes' | 'hours' | 'days' | 'weeks' | 'years';

INT: [0-9]+;
```

Figure 3. Excerpt from the timestamp grammar code

The g4 file contains the grammar code, including its structural and semantic rules (Fig. 4). ANTLR4 generates the AST as a tree structure [12] and exports it as Python code. Python then executes, analyzes, and detects vulnerabilities in the smart contract.

A parse tree is an ordered tree that represents the syntactic structure of source code. The root of the parse tree corresponds to the starting symbol of the grammar. Different parser generator applications exist for various programming languages to create parse trees.

For our approach, we use the parser Generator like the command ***$ ANTLR4 -Dlanguage=Python3 g.g4*** which can generate the parser and the token from the g4 grammar. The "Dlanguage=?" part means the programming language that we will use to detect the bug. Fig. 4 illustrates example grammar rules.

```

grammar Solidity;

sourceUnit
  : (pragmaDirective | importDirective | contractDefinition)* EOF ;

pragmaDirective
  : 'pragma' pragmaName pragmaValue ';' ;

pragmaName
  : identifier ;

pragmaValue
  : version | expression ;

version
  : versionConstraint versionConstraint? ;

versionOperator
  : '^' | '~' | '>=' | '>' | '<' | '<=' | '=' ;
```

Figure 4. Excerpt from grammatical rule .g4

After executing the command to transform the grammar rule into parser and Lexer, we automatically obtain the files illustrated in Fig. 5; if the grammatical rule had no error.

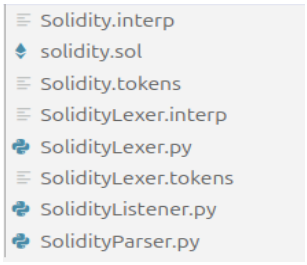

Figure 5. Files obtained

For step II', we used SmartBugs as our dataset. This dataset can be obtained from [18]. SmartBugs is a collection of smart contracts containing bugs. Etherscan, the official site [19], was used to retrieve functional smart contracts. Fig. 6 shows the contents of SmartBugs.

```
h@h:~/sol/syntax/dataset_izy$ ls -l
total 40
drwxrwxr-x 2 h h 4096 Mar 15 22:31 access_control
drwxrwxr-x 2 h h 4096 Mar 15 22:31 arithmetic
drwxrwxr-x 2 h h 4096 Mar 15 22:31 bad_randomness
drwxrwxr-x 2 h h 4096 Mar 15 22:31 denial_of_service
drwxrwxr-x 2 h h 4096 Mar 15 22:31 front_running
drwxrwxr-x 2 h h 4096 Mar 15 22:31 other
drwxrwxr-x 2 h h 4096 Mar 15 22:31 reentrancy
drwxrwxr-x 2 h h 4096 Mar 15 22:31 short_addresses
drwxrwxr-x 2 h h 4096 Mar 15 22:31 time_manipulation
drwxrwxr-x 2 h h 4096 Mar 15 22:31 unchecked_low_level_calls
h@h:~/sol/syntax/dataset_izy$
```

Figure 6. SmartBugs Contents

In Step III, the Python file (with a .py extension) calls the lexer, parser, and listener files to identify vulnerabilities. This file can also display any errors found in a smart contract, as well as the results obtained after vulnerability detection. Fig. 7 shows the python script part that will retrieve the solidity code and display the result after vulnerability detection.

```
from antlr4 import *
from SolidityvulnerabilityLexer import SolidityvulnerabilityLexer
from SolidityvulnerabilityParser import SolidityvulnerabilityParser

def main():
    # Solidity code containing timestamps
    solidity_code = '''

    '''
    # Create a lexer and parser
    lexer = SolidityvulnerabilityLexer(InputStream(solidity_code))
    stream = CommonTokenStream(lexer)
    parser = SolidityvulnerabilityParser(stream)
    # Parse the input
    tree = parser.sourceUnit()
    # Create a listener
    listener = SolidityTimestampListener()

    # Traverse the parse tree to find vulnerabiliyt
    walker = ParseTreeWalker()
    walker.walk(listener, tree)
```

Figure 7. Extract from the code to detect vulnerabilities

B. *Performance evaluation*

In this section, we evaluate the effectiveness of our approach using the FDR and FNR metrics [11]. These metrics serve to validate the performance and reliability of the approach.

Our evaluation centers on the use of a specific dataset, which is also employed by MSmart during its testing phase. The SmartBugs dataset is a collection of solidity files containing bugs or vulnerabilities in the code. It comprises a total of 143 files across 10 different file types. Using SmartBugs is essential for testing the performance, feasibility, and independence of a tool, as well as for obtaining solidity source code samples without bugs. For further reference, the official Etherscan website provides access to the public blockchain.

As MSmart already mentioned in [11], the goal is to compare our result against their result. After testing our

approach with the dataset used by MSmart [11] from the different parameters, we obtain the results shown in Table II.

We calculate the accuracy of our approach using the FDR metric, based on TP (True Positive) and FP (False Positive). Several researchers have previously applied static analysis methods, as seen in [6] and [11]. Many tools have been developed using this approach, including Securify, Slither, Mythril, Oyente, and SmartCheck [11]. (Fei et al., 2023) developed a test on their approaches [11]. The FP and FN rates are used to evaluate the performance of an approach [3, 6, 11]. A FP occurs when a vulnerability is detected in a smart contract, but in reality, the bug does not exist [3]. Conversely, a FN indicates that no bugs or vulnerabilities are detected in a contract, although the bug is actually present [3].

To calculate the accuracy rate of a bug analysis tool compared to other analysis tools; we use two metrics:

$$\mathrm{FDR} = \frac{\mathrm{FP}}{(\mathrm{TP}+\mathrm{FP})} \quad (1)$$

$$\mathrm{FNR} = \frac{\mathrm{FN}}{(\mathrm{TP}+\mathrm{FN})} \quad (2)$$

The FDR (1) and FNR (2) are proportional measures and should be expressed as percentages (%). If the FDR (1) percentage is higher compared to other tools, the tool can be considered efficient [11]. However, the FNR (2) value should be lower than that of other tools [11]. To enhance the efficiency and effectiveness of detecting bugs and vulnerabilities in the smart contract, the g4 grammar section needs to be parameterized.

TABLE II. RESULT FROM FDR AND FNR OF SMARTCHECK, MSMART AND THE TOOL

	FDR value	*SmartCheck*	*MSmart*	*Our tool*
Project 1	TP|FP|FN FDR = FP|TP+FP	1|10|5 9.1	5|10|5 33.3	6|10|4 37.5
	FNR = FN| TP+FN	83.3	50.0	40
Project 2	TP|FP|FN FDR = FP|TP+FP	2|8|6 20.0	6|8|2 42.8	7|8|2 46.6
	FNR = FN| TP+FN	75.0	25.0	22.2

Table II shows the FDR and FNR result for these three approaches. For FDR, the rate for SmartCheck increased from 9.1% in Project 1 to 20.0% in Project 2, while the rate for MSmart increased from 33.3% in Project 1 to 42.8% in Project 2. For our approach, the FDR rate reached 37.5% for Project 1 and 46.5% for Project 2. For FNR, the rate for SmartCheck decreased from 83.3% in Project 1 to 75.0% in Project 2, and the rate for MSmart decreased from 50% in Project 1 to 25.0% in Project 2. Using our approach, the FNR rate decreased to 40% in Project 1 and 22.2% in Project 2. The FDR rate of our approach was higher compared to both SmartCheck and MSmart, indicating an increase in detection accuracy. Conversely, the FNR rate of our approach decreased more significantly compared to SmartCheck and MSmart. We conclude that our approach was successful in identifying contract vulnerabilities while effectively reducing false negatives.

Table III shows the results of each SmartBugs test compared with SmartCheck and MSmart. The performance of our approach compared to other vulnerable bug detection tools.

TABLE III. COMPARISON RESULTS BETWEEN SMARTCHECK, MSMART, AND THE TOOL BASED ON SMARTBUGS DATA

Vulnerabilities and bugs SmartBugs	***Number of real bugs***	***SmartCheck***	***MSmart***	***Our tool***
IntegerOverFlow	15	0	13	13
Timestamps	5	1	2	4
Denial oS	6	0	2	5
Reentrancy	31	-	-	29
Selfdestruct (AccessC)	18	-	-	6
Deletecall (BadRando)	8	-	-	7

The source codes of the contract downloaded from Etherscan [19] have been filtered and verified. Our approach aims to reduce the number of FN and FP compared to the results of Intelligent Control.

The experimental analysis shows the following:

- Integeroverflow: Smartcheck cannot detect such bugs while MSmart and the tool can detect most integer overflow vulnerabilities. MSmart detects 13 out of 15 vulnerable smart contracts, and our tool has identified 13 types of bugs. The types of bugs our tool could not detect are "integeroverflow_mapping.sol" and "insecuretransfer.sol".
- Timestamp: SmartCheck, MSmart and the tool can detect this type of vulnerability. Our tool did not detect the file "gouvernemental_survey.sol".
- Denial oS: MSmart and the tool can detect this type of vulnerability. All 5 files were detected by our tool while the "sendloops.sol" file was not detectable.
- Reentrancy: SmartCheck and MSmart were unable to detect reentrancy-type vulnerabilities. Only our tool detected flaws on the SmartBugs, but two files are not detected such as "reetrancy.sol" and "simple_dao.sol".
- Selfdestruct and DeleteCall: we were unable to detect the type of vulnerability of selfdestruct and DeleteCall; but we detected the Access Control and BadRandom vulnerability type.

We can deduce that our approach effectively protects against the six categories of vulnerabilities. Although it cannot analyze all vulnerabilities in real-time, it is capable of detecting most of them.

The test results demonstrate that detecting vulnerabilities before deployment can be optimized by incorporating manual tests into the process. This approach and solution can be used by blockchain developers to test their smart contracts against

such types of attacks. In the state of the art discussed in related works, static testing approaches are often accompanied by false positives; adding a manual testing approach can significantly reduce these false positives.

V. Discussion

Static testing (manual testing) can detect bugs in a smart contract [10, 12]. The proposed approach improves the detection of bugs in smart contracts. However, a potential limitation of this solution is that static testing can be time-consuming and may not reliably detect issues in complex contracts. To enhance the security of smart contracts with complex architectures, it is recommended to develop two tools. The first tool should analyze solidity code broadly, covering lexical, syntactic, and semantic aspects. The second tool should focus on detecting specific vulnerabilities. We found that some vulnerabilities identified by SmartBugs were caused by certain unexpected solidity syntax elements, such as special characters and logo comments. Additionally, it is important to stay informed about the functionality of each new solidity version. While no tool can guarantee detection of all vulnerabilities, we'll have to optimize and maximize the identification of potential security issues in smart contracts over time.

VI. Conclusion and Perspectives

In this article, we presented a static analysis approach to detect and correct vulnerabilities in solidity-written smart contracts. This study proposes a methodology for identifying syntactic bugs in solidity code using grammatical rules. Our approach was to update the grammar rules already in use to support vulnerability detection with the 0.8.* version of solidity. Features such as timestamp, reentrancy, unchecked, try/catch and abicoder have been integrated into our grammar rules.

The approach was based on four main steps: (Step I) Grammatical of rule Generation to analyze the solidity source code, (Step II) Using ANTLR to analyze the solidity syntax, (Step II') Inserting the solidity code from the SmartBugs dataset to test the effectiveness of the model, (Step III) Vulnerability Detection using lexer, token and parser. The two results obtained demonstrate the particularities of our approach. The first result is the improvement of the FDR rate of 38.05% with MSmart; but thanks to our approach, this FDR rate becomes 42.05%. And the FNR rate is 37.5% with MSmart; but this FNR rate becomes 31.1% with our approach. The second result is about bug type detection in solidity. Our approach has demonstrated effective ability to identify bugs such as IntegerOverflow, Timestamps, DoS, and Reentrancy, although some vulnerabilities; notably, Selfdestruct and DelegateCall were not detected. The analysis of smart contracts from SmartBugs confirmed the effectiveness of our approach by highlighting the importance of static analysis.

However, our approach has some limitations regarding the detection of any kind of bug. The approach will not be able to analyze some types of complex vulnerabilities. The perspectives of our work mainly consist in proposing other specialized approaches for specific vulnerabilities such as reentrancy, gas-related problems and Access Control flaws and so on. In addition, the integration of Artificial Intelligence techniques will enrich the analysis and increase the accuracy of detection. These improvements will help strengthen the security of smart contracts and provide a more powerful tool for blockchain developers.

References

[1] W. Deng, H. Wei, T. Huang, C. Cao, Y. Peng and X. Hu, "Smart contract vulnerability detection based on deep learning and multimodal decision fusion," Sensors, vol. 23, no 16, p. 7246, 2023.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[3] J. Sui, L. Chu and H. Bao, "An opcode-based vulnerability detection of smart contracts," Applied Sciences, vol. 13, no 13, p. 7721, 2023.

[4] D. Han, Q. Li, L. Zhang and T. Xu, "A smart contract vulnerability detection model based on syntactic and semantic fusion learning," Wireless Communications and Mobile Computing, vol. 2023, no 1, p. 9212269, 2023.

[5] F. Ullah and F. Al-Turjman, "A conceptual framework for blockchain smart contract adoption to manage real estate deals in smart cities," Neural Computing and Applications, vol. 35, no 7, p. 5033-5054, 2023.

[6] S. Tikhomirov, E. Voskresenskaya, 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, p. 9-16, 2018

[7] M. Sharma, "Optimizing Detection of Reentrancy attacks in Solidity Smart Contracts," 2023, Thèse de doctorat. Dublin, National College of Ireland.

[8] N. Szabo, "Smart contracts: Building blocks for digital markets. 1996," URL: http://www. fon. hum. uva. nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo. best. vwh. net/smart_ contracts_2. html (дата обращения: 30 июля 2018 г.), 2017.

[9] Z. Zheng, N. Zhang, J. Su, Z. Zhong, M. Ye and J. Chen, "Turn the rudder: A beacon of reentrancy detection for smart contracts on Ethereum," in 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, p. 295-306, 2023.

[10] A. A. E. Ghaleb, "Static analysis approaches for finding vulnerabilities in smart contracts," 2023, Thèse de doctorat. University of British Columbia.

[11] J. Fei, X. Chen and X. Zhao, "MSmart: Smart contract vulnerability analysis and improved strategies based on SmartCheck," Applied Sciences, vol. 13, no 3, p. 1733, 2023.

[12] S. Kushwaha, S. Joshi, D. Singh, M. Kaur and H. N. Lee, "Ethereum smart contract analysis tools: A systematic review," IEEE Access, vol. 10, p. 57037-57062, 2022.

[13] NCC Group, "Decentralized Application Security Project (or DASP) Top 10," 2025, [Online]. Available: https://dasp.co/

[14] F. A. Alaba, H. A. Sulaimon, M. I. Marisa and O. Najeem, "Smart contracts security application and challenges: A review," Cloud Computing and Data Science, p. 15-41, 2024.

[15] B. Bellaj, A. Ouaddah, N. Crespi, A. Mezrioui and E. Bertin, "A transpilation-based approach to writing secure access control smart contracts," in: 2023 5th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS). IEEE, p. 1-7, 2023.

[16] G. A. Pierro and R. Tonelli, "Paso: A web-based parser for solidity language analysis," in: 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE). IEEE, p. 16-21, 2020.

[17] Z. H. Randriamiarison, "The detect syntax solidity," 2025, URL: https://github.com/RANDRIAMIARISON/detect_syntax_solidity.

[18] GitHub, "SmartBugs: A Framework to Analyze Ethereum Smart Contracts," 2025, URL: https://github.com/smartbugs/.

[19] Etherscan, "The Ethereum Blockchain Explorer," 2025, URL: https://etherscan.io/.