SolDetector: Detect Defects Based on Knowledge Graph of Solidity Smart Contract

Tianyuan Hu, Zhenyu Pan, Bixin Li
School of Computer Science and Engineering
Southeast University, Nanjing, China

***Abstract*—Smart contract security is one of core security issues in the application of blockchain. In recent years, attacks on smart contracts occur frequently, there are a lot of researches concerning on smart contract security issues. However, almost all solutions proposed in these researches are low precision and high False Negative Rate(FNR). In this paper, we propose a defect detection method for checking security of Solidity smart contract based on knowledge graph. Therefore, we first construct knowledge graph of smart contracts by fully integrating syntax and semantic information of Solidity source code; then, we define defect patterns by analyzing defect characteristics; furthermore, we define inference rules for defects based on knowledge graph and defect patterns; finally, we detect defects by SPARQL query. We also implement a tool named SolDetector and perform experiment on three different datasets, which shows that SolDetector is effective and efficient.**

***Index Terms*—smart contract, knowledge graph, defect detection**

I. Introduction

As a distributed public ledger technology in peer-to-peer network, blockchain provides an innovative method to store information, execute transactions, and build trust in an untrusted environment. Even though Blockchain technology provides many new mechanisms to solve security issues compared to traditional technologies. However, blockchain security is a bottleneck affecting its wide use because of the existence of vulnerabilities in smart contracts, consensus protocols, infrastructure code, etc.

Ethereum, as a representative of public chain, uses a high-level programming language called Solidity to write its smart contract. In Ethereum, a smart contract is actually a collection of codes, including various functions and various states generated by the code running process. It is compiled to Ethereum Virtual Machine (EVM) instructions for blockchain deployment. Once published on Ethereum platform, smart contracts will be executed on all nodes in the network as a program, and cannot be modified. If the deployed smart contract's code is insecure, software vulnerabilities may be exploited by malicious attackers. As smart contract code involves digital assets, it may cause huge losses once the defects of the contract code are used. Thus, how to ensure the security of the smart contract is very important.

Over the past few years, the automated analysis tools for smart contracts have made progress. Mainstream defect detection methods can be divided into static analysis [1]–[3] and dynamic analysis [4], [5]. Static analysis focuses on syntax analysis of source code, which is not suitable for complex logic analysis. For some defects with complex logic, static analysis has a low precision. Dynamic analysis has a high precision because it detects real smart contract vulnerabilities during contract executions. But dynamic analysis fails to achieve sufficient code coverage, which ignores some syntax errors and produces false negatives. So there are two open challenges in detecting smart contract.

How to increase the precision of the contract defect detection method and keep low false negatives?

How to detect more defect types of smart contract and extend the method flexibly?

In this paper, we propose a defect detection method for smart contract in Solidity based on knowledge graph to improve the precision and find more defects. We summarize our contributions as follows:

- The knowledge graph of smart contract is constructed, including the ontology layer and the instance layer.
- A defect detection method is proposed for checking security of Solidity contract based on knowledge graph, which realizes defect localization by inference rules and SPARQL.
- A tool called SolDetector is implemented to fully automate the analysis of contracts.
- An evaluation is performed to demonstrate the effectiveness and efficiency of SolDetector over three different datasets including 24,583 smart contracts.

The rest of the paper is organized as follows. Section II introduces the background knowledge of smart contract defect and knowledge graph. Section III discusses the knowledge graph construction of smart contract. Section IV details defect detection of smart contract based on knowledge graph. Section V evaluates SolDetector by experiment. Section VI discusses related work. Finally, Section VII concludes the paper and suggests future work.

II. Background knowledge

A. Smart contract defect

A contract defect is an error, flaw or fault in a smart contract that causes it to produce an incorrect or unexpected result, or to behave in unintended ways [6]. Smart contract defects are mainly caused by coding and may be related to

This paper was supported by the National Key Research and Development Program of China(No.2019YFE0105500).
DOI reference number: 10.18293/SEKE2021-133

developers, implementation language, compilers, and execution mechanism of blockchain system. We collected smart contract researches and 16 kinds of contract defects with characteristics are concluded, including Missing Reminding Execution Results defect, Balance Manipulation defect,Integer Overflow and Underflow defect, etc.

B. Defect pattern

The defect pattern is an abstract representation for defects capturing defect characteristics, including core elements, relationships between elements and restrictions on elements. A reasonable definition of defect pattern implies satisfaction of the contract defect. For example, a defect pattern of Reentrancy defect is defined as below.

1. ReentrancyPattern $\equiv$
2. $\exists$ *containAssignment*. Assignment$\sqcap$
3. $\exists$ *callFunction*. FunctionCall$\sqcap$
4. (Assignment, *follow*, FunctionCall)$\sqcap$
5. (FunctionCall, *without*, gasLimitation)

In the definition, a pattern is composed of core elements, relationships and other restrictions. Core elements in Reentrancy defect pattern are **Assignment**(line2) and **FunctionCall**(line3). The relationship between **Assignment**(line2) and **FunctionCall** is *follow*(line4). There is also a gasLimitation on **FunctionCall**(line5). A core element involved in the defect pattern can be described as a class in knowledge graph. Similarly, a relationship can be described as a object property between two classes, which restricts the logical relationship between elements. Specific defects not only have logical relationship restrictions between elements, but also have limitation on elements themselves(line5).

C. Knowledge graph

Knowledge graph is a technical to describe knowledge and construct connections between all things in the world using graph models [7]. It consists of nodes and edges. Nodes are individuals or abstract concepts. Edges are properties of individuals or relationships between individuals. Based on knowledge graph, we can identify, discover and infer complex relationships between things and concepts from data. Knowledge inference is the process of inferring unknown facts or relations based on existing facts or relations in the graph and applying certain rules to draw logical conclusions. The knowledge graph of smart contracts represents the basic syntax and semantic of smart contracts. Furthermore, more complex unknown facts that can be obtained by inferring.

III. Knowledge graph construction of smart contract

Fig.1 depicts knowledge graph construction process. Knowledge graph is constructed based on the source code of a smart contract. Combined with Solidity grammar, Abstract Syntax Tree (AST) is generated to extract information for building knowledge graph. The knowledge graph of smart contract integrates two layers: ontology layer describing abstract concepts and instance layer describing concrete facts.

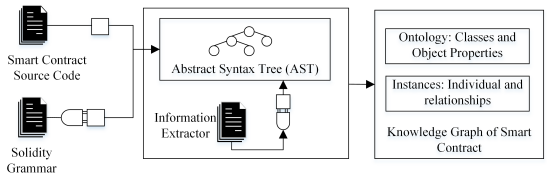

Fig. 1. Knowledge graph construction process

A. Ontology layer

Ontologies are artifacts used to model and represent knowledge related to a specific domain in an explicit way [8]. A typical ontology consists of a finite number of terms and relations between them. Terms are important concepts of the given domain. The smart contract focused in the paper is written in Solidity. Therefore, how to model and represent the Solidity by ontology is illustrated in this section.

The purpose of ontology layer is to represent Solidity in terms of concepts and relations. In this paper, the ontology layer describes code elements and corresponding relationship of Solidity source code. Code elements are modeled as classes and relationships between code elements are modeled as object properties.

For example, a Solidity code snippet is shown in Fig.2. The contract has four main code elements, including a state variable, a function, a function call and an assignment. Four main classes can be abstracted from the code snippet, including Contract, Function, StateVar, FunctionCall and Assignment. Correspondingly, relationships between classes can be concluded, such as *hasStateVar*, *containAssignment*. Hence, object properties can be abstracted as *hasFunction*, *containAssignment* and *hasStateVar*.

The illustration of the class and object property for the code snippet1 can only represent part of key elements. Full knowledge graph definition of smart contract contains 13 types of classes and 26 types of object properties.

```
1.contract SimpleWithdraw {
2.   mapping (address => uint) public credit;
3.      function withdraw(uint amount) public{
4.        require(msg.sender.call.value(amount)());
5.        credit[msg.sender]-=amount;
6.      }
7.}
```

Fig. 2. Solidity code snippet1

B. Instance layer

As the definition of classes and object properties in ontology layer, the next step is to extract required information from source code to build the instance layer and construct the entire knowledge graph. In Fig. 1, the information extractor extracts key facts as individuals and attaches relationships between individuals based on AST of source code. An abstract syntax tree parser ANTLR generates AST of smart contract source

code. Information in each AST node can be assessed by visitor. The tool can be extended to support other smart contract language by adding an ANTLR grammar. To deal with the instance extraction for multiple classes or object properties, the node visitor is defined for each class. A node visitor is responsible for the individuals generation of one class. Details of extracting individual and relationship are illustrated below with the explanation for the code snippet1.

Extracting individual. Class is an abstract concept used to describe key code elements in smart contract. Each component in source code can be extracted as an individual belonging to a class. Given the class Function in the code snippet1, it's visitor enumerates all Function and generates an individual of Function with name "withdraw".

Extracting relationship. Relationship between individuals can be extracted according to the object property definition. Relationship between different individuals can be extracted during the nested visits by visitor, which is called syntax relationships in this paper. For instance, when the visitor from the Contract node to the Function node, *hasFunction* property associates Function individual **withdraw** with it's declaring Contract **SimpleWithdraw**.

In addition, relationships include not only syntax relationship defined in smart contract, but also logical relationship. For the code snippet1, the logical relationship is currently reflected in order of execution statement within a function, as reflected by *follow*, e.g. (Assignment, *follow*, FunctionCall) (Line4 and Line3). Finally, the knowledge graph constructed for the code snippet1 is shown in the Fig. 3. More specific description are omitted for brevity.

We use OWL as an ontology description language, which is a rich vocabulary description language that can characterize relationships between classes, types of properties and characteristics of properties. The proposed method refers to OWL documents as a knowledge graph. In OWL, line numbers are stored in the individual name and help to localize defects in source code.

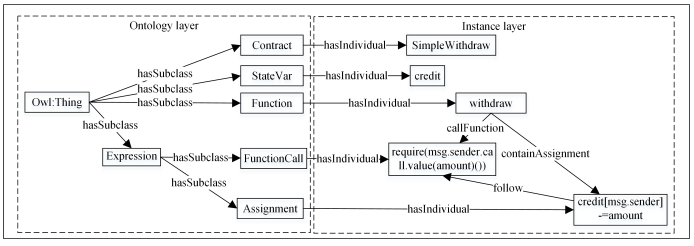

Fig. 3. The knowledge graph of the code snippet1

IV. Defect detection of smart contract

Based on knowledge graph of smart contract, we conduct defect detection process as shown in Fig.4. The process consists of two main steps: knowledge inference and defect localization.

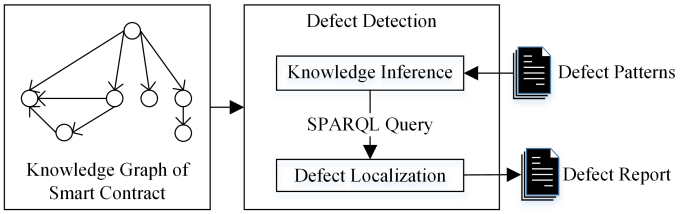

Fig. 4. Defect detection process

A. Knowledge inference

In this step, indirect knowledge is inferred upon the knowledge graph which contains facts extracted from the source code. The inference process infers the indirect relationships between code elements from known facts and relationships. For example, direct logical relationship of two adjacent code elements represented by "*follow*" in instance layer. Indirect logical relationship can be inferred from basic relationships. The facts can be expressed as triples. By matching triples, the inference rule for logical relationship can be expressed as:

$$(?A\ \textit{follow}\ ?B), (?B\ \textit{follow}\ ?C) \Rightarrow (?A\ \textit{follow}\ ?C)$$

In the rule, the variables start with a question mark (?) denote matched subjects or objects of triples [9]. The indirect logical relationship between *A* and *C* can be inferred. Similarly, the inference rule for judging types of s state variable in and arithmetic operations can be expressed below, which is critical for identifying Integer Overflows and Underflow defect.

$$(?x\ \textit{is}\ Assignment), (?x\ \textit{assignObject}\ ?y),$$

$$(?y\ \textit{typeName}\ ?type), (?type\ \textit{like}\ 'uint')$$

Inference rules are applied to capture complex relationships and abstract elements satisfying the definition of the defect pattern. Even basic fact from AST are not sufficient, complex relationships and code details critical to defect detection can be drawn by inference rules. An example that recognizes Reentrancy defect by inferring is explained below. In Fig.5, the inference rule of Reentrancy defect is shown. In contract **SimpleReentrancy**, $\textbf{Function}_{\textbf{withdraw}}$ calls a function (1) realizing transferring by "call" and contains an assignment (3) subtracting the transferring amount from **credit[msg.sender]**. According to Reentrancy defect pattern, the contract contains main code elements of Assignment and FunctionCall. Besides, $\textbf{FunctionCall}_{\textbf{require}}$ realizes transferring by "call", which has no gas limitation (2). Combined with the logical relationships inference rule, it is inferred that $\textbf{Assignment}_{\textbf{credit[msg.sender]}}$ follows $\textbf{FunctionCall}_{\textbf{require}}$ (4). Therefore, $\textbf{FunctionCall}_{\textbf{require}}$ and $\textbf{Assignment}_{\textbf{credit[msg.sender]}}$ may lead a malicious contract calls back into the contract before the first invocation of the function is finished. Because **SimpleReentrancy** contains all code elements in Reentrancy defect and meet other restrictions, the contract has Reentrancy defect.

In this section, only one example is given. For other contract defects, inference rules are customized according to defect

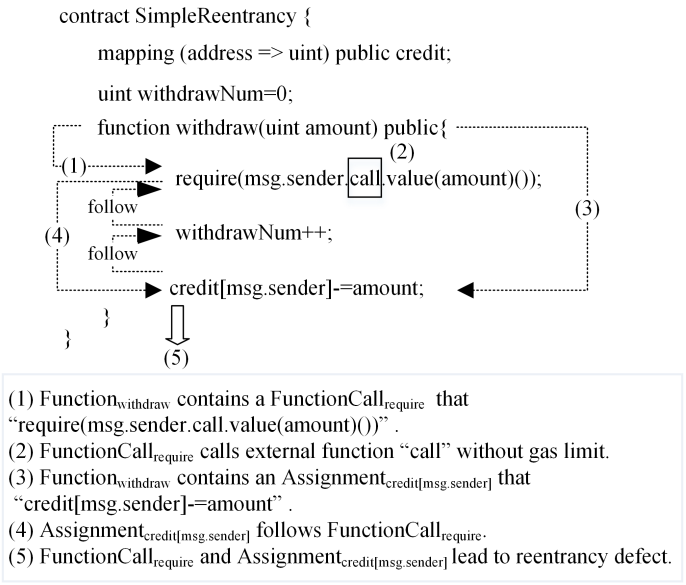

Fig. 5. Inference of Reentrancy defect

characteristics. Furthermore, inference rules are universal for smart contract in different program language.

B. Defect localization

The inference rules use SPARQL language to query and manipulate the knowledge graph data. Based on all available definition in ontology layer and instance layer, the SPARQL of a defect is able to utilize main elements and restrictions. From the perspective of defect matching, SPARQL enables flexible search strategies based on knowledge expressed in higher levels. Based on triple matching, SPARQL query is conducted on knowledge graph. Successful triple matching will return an individual name, which shows a defect is found; otherwise, no defect is found. Due to the line number is stored in each individual name, there are following two cases of successful SPARQL query.

(1) Case1: A defect pattern is satisfied (e.g., Integer Overflow and Underflow defect) and an individual name containing specific location is returned, which indicates the defective statement.

(2) Case2: A defect pattern is satisfied (e.g., Frozen Ether defect) and the contract individual name is returned. Thus, the contract has this defect and cannot be localized to specific line.

For example, one challenge in identifying Integer Overflows and Underflow defect is judging the type of an operation variable in an assignment. We illustrate the SPARQL query of Integer Overflow and Underflow defect as shown in Fig.6.

SPARQL simplifies the access to knowledge graph data to with strong expressiveness and speed optimization. Furthermore, it provides a unified interface for the management of knowledge graph data. Compared to hard-coded algorithms, SPARQL supports flexible defect detection strategies. SPARQL can be added based on available vocabularies according to the inference rule for new contract defect.

query.png

```
1.  SELECT ?subject
2.  WHERE {
3.  {?subject a sol:Assignment.
4.  ?subject sol:assignObject ?object; sol:assignValue ?value.
5.  FILTER (regex(?value,"\\+|\\-|\\*|\\/|\\^")&&regex(?object, '\\['))
6.  ?stateVar a sol:StateVar; sol:nameIs ?stateVarName;sol:typeName ?typeName.
7.  ?object sol:elementTypeName2 ?mappingType2.
8.   FILTER (regex(str(?mappingType2), "int")
10.  &&regex(?typeName, "mapping")
11.  &&regex(?object , ?stateVarName))}
14.  }
```

Fig. 6. SPARQL query of Integer Overflow and Underflow defect

V. Evaluation

To put the proposed method into practice, we have implemented SolDetector based on Jena. It integrates the Antlr generating AST and information extractor building knowledge graph for smart contract dynamically. The defect inference rules are executed by SPARQL query engine on OWL files.

We use three datasets to evaluate SolDetector. Dataset1 consists of 179 smart contracts selected from test datasets of three popular analysis tools [1], [2], [10] and has been attacked in a real-world, totaling 31,904 lines of the code. Dataset2 consists of 15,623 smart contracts crawled from Etherscan in 2018, totaling 4,197,965 lines of the code. Dataset3 consists of 8,781 smart contracts crawled from Etherscan in 2020, totaling 5,215,734 lines of the code. There is no restriction on Solidity contract versions and contracts' lines. It is useful to estimate the efficiency of SolDetector for massive contracts with different Solidity version.

A. Effectiveness of SolDetector

To evaluate the effectiveness of defect detection, we run SolDetector on three datasets and use measurements listed below. 1) TP indicates the number of vulnerable contracts detected by the tool correctly. 2) FP indicates the number of vulnerable contracts detected by the tool incorrectly. 3) TN indicates the number of contracts without defects detected by the tool correctly. 4) FN indicates the number of contracts free of defect detected by the tool incorrectly. Precision, Recall, FDR and FNR can be calculated as: $Precision = TP/(TP+FP) \times 100\%$, $Recall = TP/(TP+FN) \times 100\%$, $FDR = FP/(TP+FP) \times 100\%$, $FNR = FN/(TP+FN) \times 100\%$.

Dataset1 contains famous vulnerable contracts with distinct defect characteristics, While Dataset2 and Dataset3 consist of contracts randomly crawled. In order to simplify the evaluation, the careful analysis is mainly aimed at Dataset1 and 10% contracts of Dataset2 and Dataset3 were randomly selected to calculate TP, TN, FP, FN. Evaluation of SolDetector is shown in Table I.

For Dataset1, SolDetector successfully detected 125 real vulnerable contracts. Meanwhile, it mistakes 4 safe contracts as vulnerable. The number of vulnerable contracts omitted by SolDetector is 12. We manually analyze 12 vulnerable contracts omitted by the tool. There are 7 contracts containing Reentrancy, 2 contracts containing Balance Manipulation, 1

TABLE I
EVALUATION OF SOLDETECTOR ON THREE DATASETS

Dataset	Selected contract number	Vulnerable contract number	TP	TN	FP	FN	Precision	Recall	FDR	FNR
Dataset1	179	137	125	38	4	12	96.90%	91.24%	3.10%	8.76%
Dataset2	1562	1285	1275	5	272	10	82.42%	99.22%	17.58%	0.78%
Dataset3	878	658	655	11	209	3	75.81%	99.54%	24.19%	0.46%

contract containing Unprotected Suicide, 1 contract containing Useless Code, and 1 contract containing Erroneous Constructor Name. The FNR of Reentrancy defect is the highest, which is mainly resulted from the diverse code forms of Reentrancy defect. Although we have abstracted the inference rule for Reentrancy defect, the defect is manifested in various forms of code. Thus, only one SPARQL query is not enough. We can fix this problem by adding new inference rules for various forms of Reentrancy defect according to each different defective contract code. Similarly, the inference rules of Useless Code defect are diverse. We just abstract one situation that the assignment containing the useless operation "==", missing unused variables and other forms. More inference rules need to be enriched. Two contracts containing Balance Manipulation defect are not detected. The main reason is that some expressions do not conform to the conventional code form of this defect, such as "If(0!=this.balance)". In general, most of the false negative can be solved by expanding the inference rules and the SPARQL query.

For Dataset2 and Dataset3, the high Recall is mainly due to the fact that randomly crawled contracts may have no transaction and Ether and contain a large number of duplicate codes. Moreover, a contract may contain multiple defects. Only if all defects within a contract are detected, the detection is considered successful, which leads to the low Precision.

B. Efficiency of SolDetector

To evaluate the efficiency of SolDetector, we recorded time consumed for the experiment on three datasets. As shown in Table II, the time cost of SolDetector is mostly for the construction of the knowledge graph and the defect location. It is dependent on the size of the contract code and the defect number. It took on average 0.15s, 0.17s and 0.23s to check a single smart contract by using SolDetector on Dataset1, Dataset2 and Dataset3 respectively. For knowledge graph construction, analyzing each smart contract only cost 0.04s, 0.04s and 0.05s respectively. For 16 kinds of defects detection, defecting each smart contract cost 0.11s, 0.13s and 0.18s respectively. Because the average lines of smart contract in Dataset3 is larger than that in Dataset2, so the average time of construction and detection for Dataset3 is larger.

C. Comparison experiment

We conduct comparison with SmartCheck (SC) to demonstrate the advantages and disadvantages of SolDetector (SD). SmartCheck is an extensive static analysis tool working on Solidity source code. The empirical research [11] shows that SmartCheck tool is statistically more effective than Securify [10], Oyente [12] and Mythril [5]. Hence, we compare SolDetector with SmartCheck. We do not consider tools based on dynamic analysis, such as ContractFuzzer.

Since it is too expensive to run SmartCheck on all 20k+ contracts, we only run it on the Dataset1 that is manually annotated with defects. For a fair comparison, we focus our evaluation exclusively on 8 kinds of defects that can be detected by both tools, including Missing Reminding Execution Results(D1), DelegateCall(D2), Frozen Ether(D3), Missing Return Statement(D4), Dependency of Timestamp(D5), Unchecked Send(D6), Balance Manipulation(D7), Tx-Origin(D8). The comparison of detection result is shown in Fig.7.

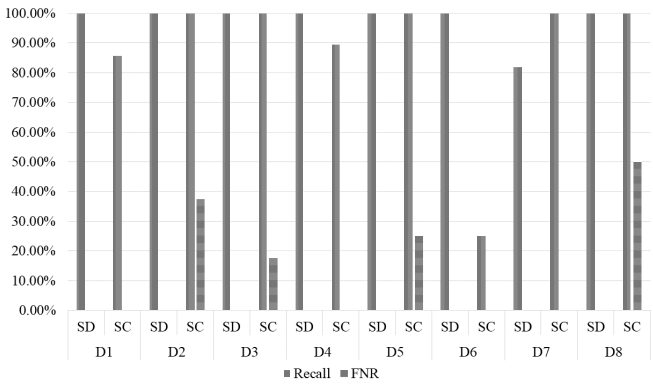

Fig. 7. Comparison of detection result between SD and SC

The following result can be drawn: 1) for seven defects, Recall of SolDetector is greater than or equal to SmartCheck. 2) for eight defects, FNR of SolDetector is less than or equal to SmartCheck.3) for only one defect (D7), SolDetector has a lower Recall of 81.82%. In general, SolDetector has a higher Recall and lower FNR, which shows the advantage of our method in detecting defects.

Furthermore, we also compare SolDetector with other tools to demonstrate it's efficiency. Tool's efficiency is related to the defect number that can be detected and the contract size. Efficiency information are collected from corresponding papers. SolDetector can detect more defects than Securify [10] and SmartShield [13]. Moreover, Securify and SmartShield cost 30s and 28s per contract, which shows that both of them are inefficient. SmartCheck is suitable for more defects and lager contract, whose average time is 1.66s per contract that is longer than SolDetector's average time. Thus, SolDetector is the fastest tool, followed by SmartCheck, Securify and SmartShield.

TABLE II
TIME CONSUMPTION OF SOLDETECTOR

	Dataset1	Dataset2	Dataset3
Contract number	179	15623	8781
Average lines	178 lines per contract	268 lines per contract	594 lines per contract
Construction time	7s	571s	466s
Average construction time	0.04s per contract	0.04s per contract	0.05s per contract
Detection time	19s	2035s	1507s
Average detection time	0.11s per contract	0.13s per contract	0.18s per contract

VI. RELATED WORK

As a distributed public ledger technology in peer-to-peer networks, blockchain is increasingly used in various fields. However, there are still security issues in smart contracts, which affects further promotion of blockchain technology. It is necessary to fully analyze potential security threats to avoid defects as much as possible. At present, the existing smart contract defect detection methods focus on symbolic execution, model checking, fuzzy testing and other methods.

Oyente [12] is the first tool to detect security problems of Ethereum smart contract. It builds control flow graph from bytecode to check whether there is any vulnerable pattern in the contract. By analyzing dependency diagram of the contract, Securify [10] deduces exact semantic information from the code. It combines compliance patterns and violation patterns constructed by semantic facts to localize contract defects. ZEUS [3] combines abstract interpretation and symbolic execution to model contract. While ZEUS does at LLVM intermediate level and cannot determine the exact location. SmartCheck [1] translates Solidity source code into an XML-based intermediate representation and checks it against XPath patterns. sFuzz [4] complements existing testing engines based on symbolic execution like Oyente [12] and Teether [14].

SolDetector makes up the shortcomings of static analysis and dynamic analysis. Knowledge graph construction for smart contract can be easily modified according to detection needs for any new code elements. Even though Solidity grammar updates and new code element are added, it is easier to extract information by generating AST with our customized Solidity grammar. Moreover, new defect patterns and corresponding inference rules can be flexibly expanded. A detection method suitable for more known defects and can be extended flexibly for new defects is significant.

VII. CONCLUSION

In this paper, we propose SolDetector, a tool for smart contract defect detection. SolDetector fully integrates syntax and semantic information of smart contracts to construct knowledge graphs by a personalized pluggable information extractors. Smart contract will be scanned for 16 kinds defect by inference rules and corresponding SPARQL, which realizes the defect localization efficiently.

Our method cannot analyze the contract execution state, which leads to a high FNR for Reentrancy defect. Combining static with dynamic analysis might be a potential way to address the disadvantage. The method proposed in this paper provides a sound basic for the combination of static and dynamic analysis. In future work, we will track and analyze the execution information to enrich the knowledge graph of smart contracts.

REFERENCES

[1] S.Tikhomirov, E.Voskresenskaya, I.Ivanitskiy, R.Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[2] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2019, pp. 8–15.

[3] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Network and Distributed System Security Symposium*, 2018, pp. 18–33.

[4] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "Sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 778–788.

[5] Mythril: An open-source security analysis tool for ethereum smart contracts. [Online]. Available: https://github. com/trailofbits/manticore

[6] J. Chen, X. Xia, D. Lo, J. Grundy, and T. Chen, "Defining smart contract defects on ethereum," *IEEE Transactions on Software Engineering*, 2020.

[7] N. Guarino, "Formal ontology, conceptual analysis and knowledge representation," *International Journal of Human-Computer Studies*, vol. 43, no. 5-6, pp. 625–640, 1995.

[8] M. Savic, G. Rakic, Z. Budimac, and M. Ivanovic, "A language-independent approach to the extraction of dependencies between source code entities," *Information and Software Technology*, vol. 56, no. 10, pp. 1268–1288, 2014.

[9] R. Xiong and B. Li, "Accurate design pattern detection based on idiomatic implementation matching in java language context," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering*, 2019, pp. 163–174.

[10] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.

[11] R. M. Parizi, A. Dehghantanha, K. K. R. Choo, and A. Singh, "Empirical vulnerability analysis of automated smart contracts security testing on blockchains," in *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, 2018, pp. 103–113.

[12] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[13] Y. Zhang, S. Ma, J. Li, K. Li, and D. Gu, "Smartshield: Automatic smart contract protection made easy," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering*, 2020, pp. 23–34.

[14] J. Krupp and C. Rossow, "Teether: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium*, 2018, pp. 1317–1333.