

SmartGraph: Static Analysis Tool for Solidity Smart Contracts

Andrei Zhukov and Vladimir Korkhov(✉)

Saint-Petersburg State University, 7-9 Universitetskaya Embankment, St. Petersburg 199034, Russia
st076077@student.spbu.ru, v.korkhov@spbu.ru

Abstract. Smart contracts written in Solidity can have various vulnerabilities, which can lead to significant losses if not detected and addressed in time. Static analysis is an effective method to identify potential security issues before deploying smart contracts on the blockchain.

In this paper, we present a tool for static analysis of Solidity smart contracts that uses a graph as an intermediate representation to facilitate analysis. The tool converts Solidity code into a graph structure that captures the code's structure and dependencies, allowing analysis at a higher level of abstraction.

The tool provides a set of rules for detecting various security issues such as reentrancy attacks, integer overflows, and state variables shadowing. We evaluate the tool on a set of Solidity contracts, present en experimental comparison of the tool performance compared to other existing tools, and demonstrate its effectiveness in detecting security vulnerabilities previously reported in the literature.

The tool's use of a graph-based intermediate representation enables a high level of abstraction that can capture complex relationships and dependencies between the various components of the code. Overall, the proposed tool provides an efficient and effective approach to perform static analysis of Solidity smart contracts.

Keywords: Solidity · Static analysis · Graph representation · Security vulnerabilities

1 Introduction

Currently, the blockchain technology is gradually being implemented in many systems as it ensures data security and integrity. Each blockchain represents a network of nodes that collectively control data movement and verification in the network. Each operation performed by a node is coordinated with the other nodes to reach consensus. Once agreed upon, the data is recorded in a registry, a copy of which is maintained by each node. Blockchain is applied to solve a large number of problems.

One of the important technologies, Merkle Tree [2], on which blockchain is based, was introduced in 1979. This data structure is used for secure and efficient

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
O. Gervasi et al. (Eds.): ICCSA 2023 Workshops, LNCS 14104, pp. 584–598, 2023.
https://doi.org/10.1007/978-3-031-37105-9_39

data encryption. In 1982 [3], a system of repositories for creating and maintaining computer systems was introduced, which were shared among several organizations to support collaboration between them without centralized services. Each of these repositories signed, recorded, and broadcasted transactions to all nodes. In 1991 [4], an article was published on establishing timestamps for digital documents. This article proposed a solution to prevent backdating digital documents. The goal was to maintain complete document privacy without a centralized service.

After the financial crisis in 2008 [1], which was associated with the collapse of various financial institutions such as Lehman Brothers, a scientific paper was published by Satoshi Nakamoto on the Proof of Work algorithm with a description of a new electronic currency based on a cryptographic protocol for secure transactions that solves the problem of double spending. The Bitcoin system was developed in 2008, and the first bitcoin was created on January 3, 2009.

However, to create a new cryptocurrency, there were two ways [5]: to create a new independent network or to create a protocol over the Bitcoin network. Creating an independent network entails creating and testing infrastructure, which may be excessive for small applications. Creating a protocol over the Bitcoin network can also be challenging because the created protocol will not inherit Bitcoin's simplified transaction verification scheme. Furthermore, the scripting language used in the Bitcoin network has its drawbacks: it is not Turing complete (lacks loops), lacks state, lacks access to nonce, and the hash of the previous block. To address the issues of creating blockchain applications, integrating autonomous decentralized services with each other, and using a cryptocurrency different from Bitcoin, the decentralized platform Ethereum emerged in 2015, along with the Solidity language and smart contracts.

Companies appear daily that implement their ideas using blockchain technology. Furthermore, projects are being developed around the world to solve problems related to data integrity and resistance to physical intervention using blockchain technology at the state level. An example of this is CBDC (Central Bank Digital Currency), which is being promoted by the US Federal Reserve System. Therefore, given the current pace of development of various decentralized services, there is a need for writing high-quality code. This is especially important for companies providing such services, as publishing a smart contract costs money. Therefore, by increasing the number of bugs found during the development phase, these companies can save money and avoid losing users due to critical malfunctions. To improve the quality of the code, developers can use various tools for visualization, testing, and analysis. However, only testing and static analysis can work completely independently.

In modern information systems, static analysis can be used in conjunction with unit, e2e, and integration testing when developers update their codebase. This is because static analysis tools can improve code quality, avoid vulnerabilities, and even adhere to a unified coding style. This is particularly important when developing smart contracts, as they involve working with finances. There-

fore, a developer's mistake can lead to the loss of control over the smart contract and users' funds.

Solidity [9] has emerged as the most widely used programming language for writing smart contracts on blockchain platforms such as Ethereum. This can be attributed to its maturity and wide adoption, as it has been around longer and has a larger community of developers and users than other languages. This popularity has resulted in a vast ecosystem of resources, tools, and libraries being developed for Solidity, making it easier for developers to build and maintain smart contracts.

Another significant factor contributing to Solidity's appeal is its flexibility. It offers a wider range of features and capabilities than other languages, making it more suitable for developing complex smart contracts. Solidity's support for inheritance, interfaces, and libraries enables developers to reuse code and create modular designs, which improves code maintainability and reduces the risk of errors. Additionally, Solidity's syntax is similar to that of popular programming languages such as C++ and JavaScript, which reduces the learning curve for developers. Given the popularity and flexibility of Solidity, it makes sense to design SmartGraph for analyzing Solidity code.

The paper is structured as follows: Sect. 2 introduces related works; Sect. 3 gives an overview of possible vulnerabilities in smart contracts; Sect. 4 considers approaches to static analysis of smart contracts; Sect. 5 presents the architecture and concepts of the SmartGraph tool, its implementation and evaluation compared to other tools; Sect. 6 concludes the paper.

2 Related Works

Slither [6] is a static analysis framework for smart contracts that can identify security vulnerabilities and coding errors. It can analyze Solidity and Vyper contracts and generate reports highlighting potential issues, such as reentrancy vulnerabilities,uncontrolled selfdestruct, and unprotected functions. Slither can also be extended with custom rules to detect specific patterns or behaviors in contracts.

One of the features that sets Slither apart from other smart contract analysis tools is its SlithIR intermediate representation. SlithIR is a custom intermediate language that is used to represent the contracts being analyzed. This allows Slither to perform more advanced analysis techniques, such as taint analysis, which tracks the flow of data through a contract to identify potential security risks. SlithIR also enables more precise detection of issues, as it can identify issues even if the code has been obfuscated or if it is written in a non-standard way.

SmartCheck [7] is a tool for detecting vulnerabilities in smart contracts written in Solidity. It uses symbolic execution to systematically explore all possible execution paths of a contract and identify vulnerabilities, such as reentrancy or arithmetic overflows. SmartCheck also provides a report with details on the vulnerabilities found, including the exact lines of code where they occur and

potential fixes. The tool has been used to analyze thousands of smart contracts and has identified numerous vulnerabilities that were subsequently fixed by the contract developers.

In addition to its vulnerability detection capabilities, SmartCheck also includes features for analyzing gas usage and providing suggestions for optimizing contract code. This can be particularly helpful in reducing the cost of deploying and executing contracts on the Ethereum network. SmartCheck is an open-source tool and is available for free on GitHub, making it accessible to anyone interested in improving the security and efficiency of their smart contracts.

SmartFast [11] is a tool designed for automated security analysis of smart contracts in Ethereum. It uses symbolic execution and concolic testing techniques to detect security vulnerabilities such as reentrancy, arithmetic overflow/underflow, and gas limit vulnerabilities. SmartFast supports the analysis of both Solidity and Vyper smart contracts, and it can also be used for smart contract optimization and bug-fixing.

SmartFast uses a symbolic execution engine to explore different paths of a smart contract's execution and generate symbolic constraints for each path. These constraints are then fed into a constraint solver to generate concrete inputs that can trigger different paths and reveal vulnerabilities. SmartFast also uses a concolic testing engine to generate inputs that maximize code coverage and help in detecting edge-case vulnerabilities. Compared to other symbolic execution-based tools, SmartFast is relatively fast and can analyze large contracts in a short amount of time. However, its effectiveness heavily depends on the quality of the symbolic constraints and the scalability of the constraint solver.

The primary distinguishing feature of SmartGraph, which we introduce in this paper, is its utilization of a graph-based intermediate representation to capture complex relationships and dependencies within Solidity smart contract code. Unlike other static analysis tools, which may rely on simpler data structures, SmartGraph's use of a graph structure allows for a higher level of abstraction, enabling more comprehensive analysis of the codebase. This unique approach provides developers and auditors with a powerful tool for ensuring the security and correctness of their smart contracts, ultimately contributing to the continued development and adoption of the blockchain technology.

3 Vulnerabilities in Smart Contracts

A vulnerability is a flaw in a system that can be intentionally exploited to compromise its integrity and cause improper behavior. While the SWC Registry [14] provides a comprehensive list of known vulnerabilities in Solidity code, SmartGraph currently focuses on a specific subset of these vulnerabilities as it is still in its early stages of development. This focused approach allows SmartGraph to effectively test its ideas and ensure its tool is working as intended before expanding its capabilities to detect a broader range of security issues.

Default Function Visibility. Functions without defined visibility are public by default. The problem may arise if a malicious actor is able to alter the state of the smart contract through this function. It is recommended to choose the visibility of the function that suits its role in the smart contract to reduce the attack vector.

Variable Overflow. Numeric data types in Solidity versions before 0.8.0 could overflow. This means that a variable of type uint8, for example, will become 0 when one is added to 255. Such behavior can seriously affect the system's operation. For example, if we store a user's balance in the smart contract state, the balance may become zero again due to an addition operation when it exceeds a certain limit. Therefore, it is necessary to use the latest available versions of Solidity or use the special SafeMath library to prevent such issues.

Outdated Solidity Version. It is recommended to use the latest available version of Solidity to write smart contracts, as newer versions fix bugs that can lead to incorrect program behavior.

Unfixed Solidity Version. It is recommended to use a fixed Solidity version, such as pragma solidity 0.8.0, to prevent accidental deployment of a smart contract with an outdated Solidity version and incorrect system behavior.

Use of the Call Method Without Exception Handling. The call function allows interaction with other smart contracts, but errors that may result from its execution need to be handled. If appropriate error handling is not added when calling the call method, subsequent program logic may behave incorrectly.

Unsafe Use of the Selfdestruct Instruction. The selfdestruct function allows termination of a smart contract's operation and transfer of all stored ether to an address specified in the function's argument. The problem arises when an attacker manages to make a call to a function that contains selfdestruct, for example, if the function is public and available for anyone to call within the smart contract.

Reentrancy. One of the most popular and serious vulnerabilities for a smart contract is reentrancy, which involves calling other smart contracts with subsequent interception of the program's execution flow by these contracts. To avoid reentrancy, it is necessary to ensure that all changes to state variables are made before calling other smart contracts, or you can use a function modifier provided by the ReentrancyGuard library from OpenZeppelin.

Default Visibility of State Variables. In Solidity, the visibility of state variables, such as "public", "private", "internal" and "external", determines who can access and modify them. By default, state variables have internal visibility, which means they can be accessed by all functions within the contract, but not by other contracts or external accounts. It is crucial to set the visibility of state variables explicitly to avoid potential security issues and unexpected behavior in the smart contract. If a state variable is left with its default visibility, it may be accessed and modified by any function within the contract, even if it was not intended. This can lead to undesired changes to the state of the contract, making it vulnerable to attacks. Therefore, it is recommended to set the visibility of state variables explicitly, according to the needs of the smart contract and its functions, to ensure that only authorized parties can access and modify them.

Shadowing State Variables. It is not recommended to name variables used in functions or modifiers the same as state variables. In this case, the variable will shadow the state variable, which can lead to misunderstandings.

4 Static Analysis of Smart Contracts

Since the Ethereum network uses the Ethereum Virtual Machine (EVM) as a key component, smart contracts written in high-level languages must be compiled into EVM bytecode to run [9]. Static analysis tools for smart contracts are divided into two types [8]:

- Analysis of EVM bytecode
- Analysis of Solidity code

4.1 Approaches to Analyzing EVM Bytecode

- Using Symbolic analysis, a Control flow graph (CFG) is constructed, on which predefined rules that determine vulnerabilities are examined. Problem: the cumbersome process of constructing CFG [12] and unreadable bytecode make it difficult to form simple rules for vulnerabilities, making this approach to vulnerability detection ineffective.
- To construct an intermediate representation (IR) by extracting the semantics of bytecode and mapping instructions that may be vulnerable to predefined security patterns. Problem: Since bytecode cannot fully represent the semantics of the Solidity language, the analysis is susceptible to false positives and false negatives (FP and FN).

Therefore, analyzing Solidity code is more effective than analyzing EVM bytecode.

4.2 Approach with Solidity Code

The Solidity code analysis approach involves converting Solidity into an intermediate representation (IR) and then using pattern matching to identify vulnerabilities. This approach is adopted by various tools, including Slither and SmartCheck. However, these tools have their limitations. For instance, Slither lacks support for some basic semantics such as var variables, and SmartCheck makes it challenging to create new vulnerability detection rules.

Building IR. Static single assignment [13] (SSA) is a program representation technique used in compiler design and optimization. In SSA form, every variable is assigned exactly once, and each assignment creates a new variable. This allows for easier analysis of the program's control flow and data dependencies, which in turn enables more effective optimizations. SSA has become a popular representation for intermediate code in modern compilers and is widely used in both academia and industry (Fig. 1).

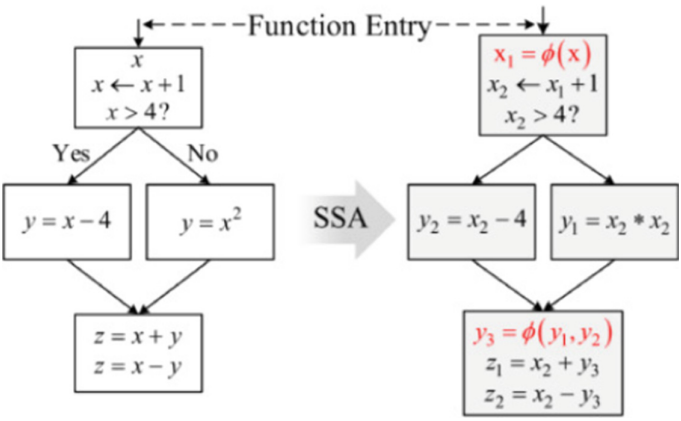


Fig. 1. Example of SSA operation.

In SSA, when a static variable x is assigned, it is represented as x1 and x2, which means that SSA requires each variable to be assigned only once. As a result of this requirement, it is easier to determine the relationships between variables, thus making data dependency analysis more efficient. When different paths are encountered in IR, such as in the case of a conditional block, SSA represents this ambiguity through the phi function, which is calculated based on the current path. If y3 is equal to y1, then z1 will be computed as x2+y1, otherwise as x2+y2.

Definition of Pattern. Often pattern matching is done through brute force search or taint analysis. For example, XPath is a method for detecting information in XML. It traverses the XML document's tags and attributes to find the information. The XML and XPath-based approach is used in SmartFast.

Taint analysis is an analysis of input data obtained from a user. Every variable that is defined by the user is considered as tainted. Every variable based on a tainted variable also gets a similar status. Programs that perform static analysis obtain a complete list of such variables and if they participate in potentially dangerous calls, such as accessing or writing to a database or working with the operating system, the programs will output an appropriate warning.

Abstract Syntax Tree. An abstract syntax tree (AST) [10] represents the structure of a program in an abstract form. The AST is created by the parser during the syntactic analysis of a program. In typical programming languages like Solidity, expressions, statements, and declarations make up the program structure.

Expressions include variable access, literals, unary and binary operators, and function calls. Programming languages offer various statements such as variable declaration, assignment, conditional statements (e.g., if), loops, control flow statements, and code blocks, which define the scope of variables.

Declarations introduce a new named entity, such as a function or a type, to the program. ASTs are useful in detecting vulnerabilities since they can capture the control flow and data flow of a program, allowing static analysis tools to identify patterns that could indicate potential security issues.

Control Flow Graph. CFG is a graph representation of all possible program execution paths. In this graph, each node represents an instruction, and each edge represents a transition from one instruction to another. Typically, the representation includes an entry node where program execution starts and an exit node where the program ends. Let's consider the property of reachability of this graph. This property is very useful for optimization. If there is a subgraph in the graph that cannot be reached from the main program execution flow, then this code is unreachable and can be removed. If the exit node is unreachable, then there may be an infinite loop in the program. Thus, we have discussed concepts that are essential for modern static code analysis, as well as approaches used by tools such as Slither.

In conclusion, our tool utilizes various techniques, such as abstract syntax tree (AST), and control flow graph (CFG), to enable comprehensive analysis of the code. The AST and CFG in our tool enable detailed analysis of the code structure and control flow, thus allowing the detection of complex vulnerabilities. The AST analysis is capable of identifying potential issues with the code's syntax, while the CFG analysis can reveal potential control flow issues. The combination of techniques used in our tool provides a comprehensive approach to identifying security vulnerabilities in smart-contracts.

5 SmartGraph: A Smart Contract Static Analysis Tool

In this section we present the design and evaluation of the SmartGraph [15], the tool based on using graphs as an intermediate representation for detecting vulnerabilities with help of static analysis of smart contracts.

5.1 Architecture

Like any static analysis tool, the smart contract code is inputted into the tool. The code is then transformed using a Solidity parser, which returns the syntax tree of the smart contract. The syntax tree describes the smart contract as a set of contract definitions along with state variables, events, modifiers, and functions defined within them. Functions, in turn, are represented as a set of expressions.

Let's consider the services that the analyzer consists of. The main service is the ParserService, which is responsible for parsing the syntax tree and constructing the Control Flow graph for functions. It can also handle variable types, various Solidity constructs such as while loops, binary operations, unary operations, accessing object properties, and so on.

The GraphService allows building an intermediate representation and using it for analysis. It supports searching for nodes of different types in the graph, adding nodes to the graph, searching for nodes by name, searching for nodes by edges, for example, to construct the Control Flow graph of a function, and searching for nodes in the graph that use state variables.

The IssueService is responsible for finding vulnerabilities in the smart contract based on the intermediate representation constructed. It represents a list of vulnerabilities from the smart contract vulnerability registry, each of which is analyzed.

The ReportService is responsible for generating a report on vulnerabilities, provides an API for adding vulnerabilities to the report, and calculates the number of vulnerabilities (Fig. 2).

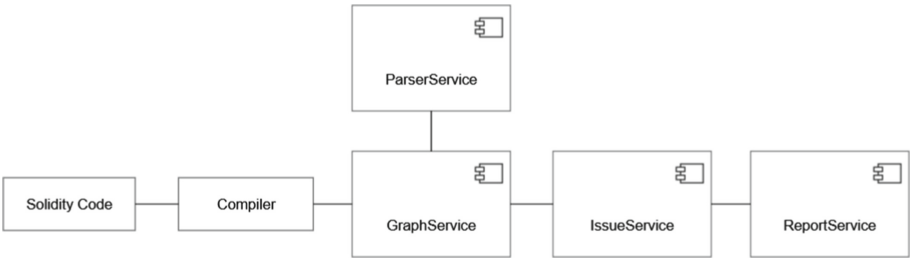


Fig. 2. Architecture.

5.2 Intermediate Representation

The model of this tool includes an intermediate representation in the form of a graph. This means that information about the smart contract is contained in

the nodes and edges of a graph. For example, we can record information about a binary operation in a node. In this case, the node will contain the identifier of the line where this binary operation is located in the smart contract, the operator that is applied to the left operand, the left and right operands, which are variables or expressions. Recording information in an edge is used only in one place in the tool: when using a state variable of the smart contract in some expression. In this case, the edge will store information about that expression. Let's examine the intermediate representation in more detail.

Nodes in Intermediate Representation. To build an intermediate representation, the tool traverses the syntax tree. Currently, the traversal goes through the definitions of smart contracts, modifier definitions, Solidity version, event definitions of the smart contract, state variable definitions, using for instructions, and function definitions. From the smart contract definition, the name of the smart contract and all contracts that this contract inherits from are extracted. Accordingly, a node is created in the graph for each definition, which we can manipulate through GraphService.

The definition of a state variable yields nodes in the graph that contain information about the variable's visibility, immutability, type, and name.

From the using for statements, information is extracted indicating the presence of connected libraries such as ReentrancyGuard or SafeMath, as well as the type of variable for which the library is applied.

Finally, a node is added for each function in the smart contract, as well as for each parameter used in that function. The node defining a function contains the following information: name, function parameters, visibility, modifiers, and the function body, which is a Control Flow graph.

Functions. Most potential vulnerabilities are found in function bodies, so their analysis needs to be approached thoroughly. However, it's also important to correctly handle the function definition to obtain important modifiers, visibility, and parameters. The function body consists of instructions, and it can be expressed through a Control Flow graph. Therefore, to construct such a graph, we need to go through each instruction and determine what it represents. For example, if the instruction is a for loop, we need to parse the variable that can be initialized in it, the condition under which it continues to run, the expression that is executed at the end of each iteration, and the body of the for loop itself. If it's an if statement, we need to parse the condition, the body for the true condition, the body for the false condition, but the latter can often be another condition for the if statement, so the process should be recursive.

It is important to note that the body of a function consists of statements that are defined through expressions. For example: address caller = msg.sender;

Here, the statement is the variable declaration, and the expression is accessing the sender property of the msg object. Parsing Solidity involves defining two main operations: parseStatement, which handles parsing statements, and parseExpression, which handles expressions, and applying these two operations

recursively, we can obtain information about arbitrarily complex statements. However, different statements can vary greatly, so for ease of working with them, the concept of order was introduced in the node. That is, each node is associated with a string of the form "parameter => use of variable." For example, consider the require(msg.value) instruction. The order property for this statement would look like msg => msg.value => require(msg.value). Thus, we can easily understand which variables are used in the statement and in what order it is executed.

5.3 IR Example

The following code provided is a simple example of a smart contract written in Solidity. The contract is called SimpleDAO and has a single function called donate that allows users to donate Ether to other addresses.

```
pragma solidity 0.4.24;

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

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

In Fig. 3, you can see the various components of the SimpleDAO contract represented as nodes in the IR. For example, the contract meta information is represented as a version node and contract name node, the mapping of addresses to credits is represented as a state variable node, and the donate function is represented as a function node.

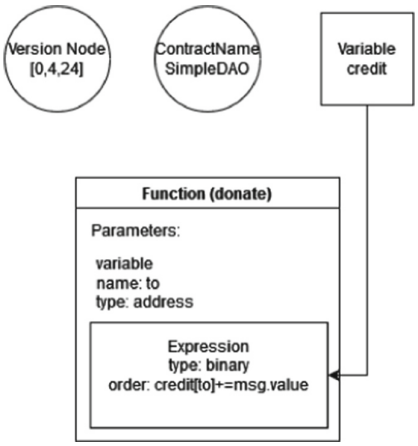


Fig. 3. SimpleDAO contract's intermediate representation.

This contract is written in Solidity, a programming language for building decentralized applications on the Ethereum blockchain. The contract is called ShadowingInFunctions and contains three functions: test1, test2, and test3. Function test3 demonstrates the concept of state variable shadowing (Fig. 4).

```
pragma solidity 0.4.24;

contract ShadowingInFunctions {
    uint n = 2;
    uint x = 3;

    function test1() constant returns (uint n) {
        return n; // Will return 0
    }

    function test2() constant returns (uint n) {
        n = 1;
        return n; // Will return 1
    }

    function test3() constant returns (uint x) {
        uint n = 4;
        return n + x; // Will return 4
    }
}
```

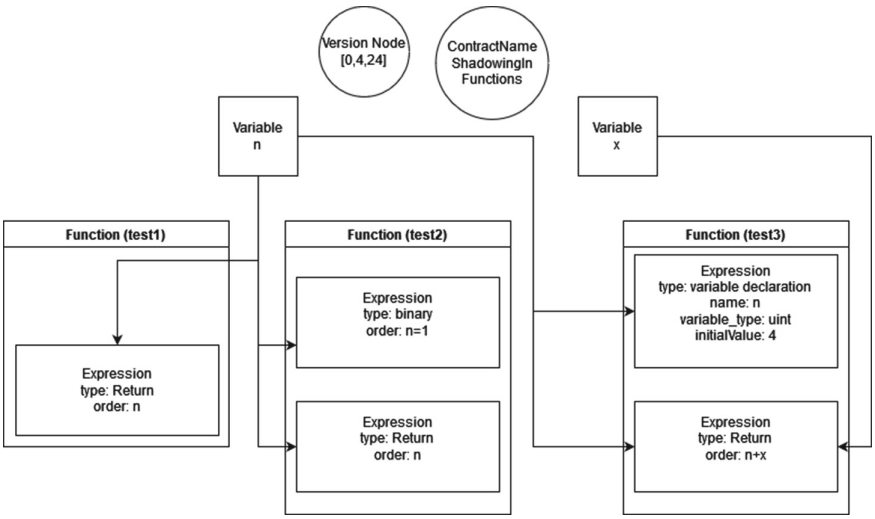


Fig. 4. ShadowingInFunctions contract's intermediate representation.

5.4 Intermediate Representation Analysis

To detect a vulnerability, you need to use the GraphService API to retrieve information about the nodes of the intermediate representation. For example, to analyze the version, you need to refer to the node of the "version" type. In turn, to detect reentrancy, you need to refer to several types of nodes: "using-for", "modifier", and "function", as information about these nodes will allow you to determine whether the state change is a vulnerability or not. Let's consider a few examples of smart contract vulnerability research:

Default Function Visibility. One of the simplest vulnerabilities to detect in this model is as follows:

1. Get all nodes whose type is function.
2. Check the visibility property of the node. If it is equal to default, then the vulnerability is detected and information about this vulnerability needs to be recorded.

Using the Call Method Without Exception Handling. Let's consider a vulnerability that is directly related to the code of the smart contract functions. In this case, it is necessary to analyze the Control Flow graph of the function for the presence of a call function and calls to the assert or require functions. These functions are used to handle exceptions.

1. Get all nodes of type function.
2. From the Control Flow graph of the function body, get all instructions.
3. For each instruction, analyze the order property.
4. If it contains a call and does not contain assert or require, then the vulnerability is detected.

Reentrancy. Let's now consider a vulnerability that relies on the edges between state variables and the instructions that use them. Thanks to these edges, we don't have to look at all the instructions, which simplifies the detection of this vulnerability and reduces the number of false positives.

1. Get all nodes whose type is function.
2. Get inheritance information through nodes of type inherit.
3. Get all instructions that use state variables, in this case, we only need the line numbers.
4. For each function, determine the presence of the nonReentrant modifier from the OpenZeppelin ReentrancyGuard library.
5. Also, check that the contract inherits from ReentrancyGuard.
6. Find all instructions that use the call function.
7. If the function does not have the nonReentrant modifier and there is an instruction that changes the state and comes after the call function call, then the vulnerability is detected.

5.5 Comparative Analysis with Other Analysis Tools

In this comparative analysis, 36 different smart contracts were used, some of which contained vulnerabilities. The smart contracts were taken from the registry of smart contract vulnerabilities [14] (Table 1).

Table 1. Comparative analysis.

Instruments	SmartGraph	Slither	SolHint	Solium
Flagged contracts	31	36	36	36
Detections per contract	3.08	6.19	5.02	4.25
False positives	7	4	0	0
True positives	8	9	4	3
False negatives	14	13	18	19
True negatives	7	10	14	14

Here it is important to understand that the tool described in this work detects only a part of vulnerabilities defined in the registry of smart contract vulnerabilities.

6 Conclusion and Future Work

In this paper, general approaches and concepts used in code analysis, as well as options for static analysis of smart contracts, were considered. SmartGraph, a new instrument for analyzing smart contracts written in Solidity based on an intermediate representation in the form of a graph was presented.

From all of the above, we can conclude that this approach allows us to describe the rules for detecting vulnerabilities quite simply and effectively. Building additional links between nodes in the intermediate representation simplifies the process of analyzing a smart contract to detect vulnerabilities.

However, from the comparative analysis, it can be seen that the next step is to refine the Solidity model since the instrument works incorrectly in 5% of cases. It is also necessary to add the ability to detect all vulnerabilities from the register of smart contract vulnerabilities and to look for places where additional links, built using edges in the graph, will allow for more efficient analysis.

References

1. Guegan, D.: The Digital World: I - Bitcoin: from history to real life. University of Ca' Foscari (2018)
2. Merkle, R.C.: Secrecy, authentication, and public key systems. Stanford University (1979)

3. Chaum, D.: Blind signatures for untraceable payments. In: Chaum, D., Rivest, R.L., Sherman, A.T. (eds.) Advances in Cryptology, pp. 199–203. Springer, Boston, MA (1983). https://doi.org/10.1007/978-1-4757-0602-4_18
4. Haber, S., Stornetta, W.S.: How to time-stamp a digital document. J. Cryptol. **3**(2), 99–111 (1991). https://doi.org/10.1007/BF00196791
5. Buterin, V.: Ethereum: a next-generation smart contract and decentralized application platform (2014)
6. Feist, J., Grieco, G., Groce, A.: Slither: a static analysis framework for smart contracts. arXiv preprint arXiv:1908.09878 (2019)
7. Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y.: SmartCheck: static analysis of Ethereum smart contracts. In: WETSEB'18: IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB 2018), 27 May 2018, Gothenburg, Sweden, p. 8. ACM, New York, NY, USA (2018)
8. Striewe, M., Goedicke, M.: A review of static analysis approaches for programming exercises. University of Duisburg-Essen (2014)
9. SolidityLang: Solidity Language Description. https://docs.soliditylang.org/en/v0.8.17/index.html. Accessed 24 Dec 2022
10. ps-group: Abstract Syntax Trees. https://ps-group.github.io/compilers/ast. Accessed 07 Apr 2023
11. Li, Z., Li, S.: SmartFast: an accurate and robust formal analysis tool for Ethereum smart contracts. Inf. Eng. Univ. (2022)
12. Allen, F.E.: Control flow analysis. IBM Corporation (1970)
13. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N.: Efficiently computing static single assignment form and the control dependance graph. IBM Research Division (1991)
14. SWC Registry: Smart Contract Weakness Classification and Test Cases. https://swcregistry.io. Accessed 16 Apr 2023
15. SmartGraph repository. https://github.com/NixoN2/smart-graph. Accessed 22 Apr 2023