Enhancing Smart Contract Security through Static Code Analysis

Dmytro Khimchenko
Czech Technical University
Ackee Blockchain Security
Prague, Czech Republic
khimchenko.dmytro.2003@gmail.com

Josef Gattermayer
Czech Technical University
Ackee Blockchain Security
Prague, Czech Republic
josef.gattermayer@ackee.xyz

***Abstract* — This article presents a novel static analysis approach for enhancing Ethereum smart contract security through the Wake framework. Our implementation focuses on detecting "write-after-write" vulnerabilities—a critical issue where variables are overwritten before their values are utilized. The approach leverages the complementary relationship between Control Flow Graphs (CFGs) and Data Dependency Graphs (DDGs), enabling precise tracking of variable operations throughout code execution paths. By correlating in-edges (write operations) and out-edges (read operations) in the DDG with corresponding nodes in the CFG, the detector achieves significantly higher accuracy than existing solutions.**

In our empirical evaluation, we compared our Wake-based detector against an equivalent implementation in Slither using both controlled test cases and real-world contracts. Results demonstrate that the Wake-based detector consistently achieves superior precision and substantially higher recall across all test scenarios. The F1-score of our implementation significantly outperforms Slither's implementation in both experimental settings. These results demonstrate how leveraging graph-based program analysis techniques with precise variable operation tracking can dramatically improve vulnerability detection in smart contract security analysis.

I. Introduction

Blockchain technology has revolutionized the way transactions are recorded and verified, with Ethereum leading the charge by introducing self-executing smart contracts with the terms of the agreement directly written in code. While smart contracts offer trustless execution, they also present unique security challenges.

Smart contracts are immutable once deployed, and any vulnerability can lead to significant financial losses, as evidenced by high-profile breaches like the DAO attack in 2016[1]. Static analysis tools have become important resources for developers to detect and mitigate vulnerabilities during the development phase.

This article discusses the importance of static analysis in the context of Ethereum smart contract security, outlines the development of a new static analysis tool, and highlights its effectiveness in identifying common vulnerabilities.

II. Background and Motivation

A. Ethereum and Smart Contracts

Ethereum extends blockchain functionality by allowing developers to write smart contracts using languages like Solidity. These contracts are compiled into bytecode and executed on the Ethereum Virtual Machine (EVM)[2]. The decentralized nature of Ethereum, combined with its consensus mechanisms, and append-only ledger structure, means that once a contract is deployed, it cannot be altered, making pre-deployment security analysis critical.

B. Security Challenges

Smart contracts have been susceptible to various vulnerabilities, including reentrancy attacks, integer overflows and underflows, and unauthorized access[3][4]. Traditional software testing methods alone are insufficient due to the complexity and unique execution environment of smart contracts, though they still play an important role in functional validation. However, there remains a significant need for specialized tools for the constraints of the EVM environment.

C. Static Analysis Tools

Static analysis involves reasoning about code without executing it, allowing developers to detect potential vulnerabilities early. Tools such as Slither[5] have been developed to automate this process for Ethereum smart contracts. However, there is still a need for more comprehensive solutions that can detect a broader range of issues in more precise ways than using heuristic methods. The Wake[6][7][8] framework is a tool that gives a developer of the static code analysis detector this opportunity. By providing a control flow graph of every function of the contract and a data dependency graph for every variable used, the developer can create detectors that are more precise and have a lower false positive rate.

III. Methodology

A. Tool Development

Our static analysis tool was developed with the goal of improving existing solutions by providing a deeper tool for analysis.

- **Language Server Support**: It is designed to analyze Solidity source code and EVM bytecode on the fly during the development phase in the IDE.
- **Analysis Techniques**: Combined abstract syntax tree (AST) analysis with control flow graph (CFG) and data dependency graph (DDG) examination to understand the contract’s structure and data flow.

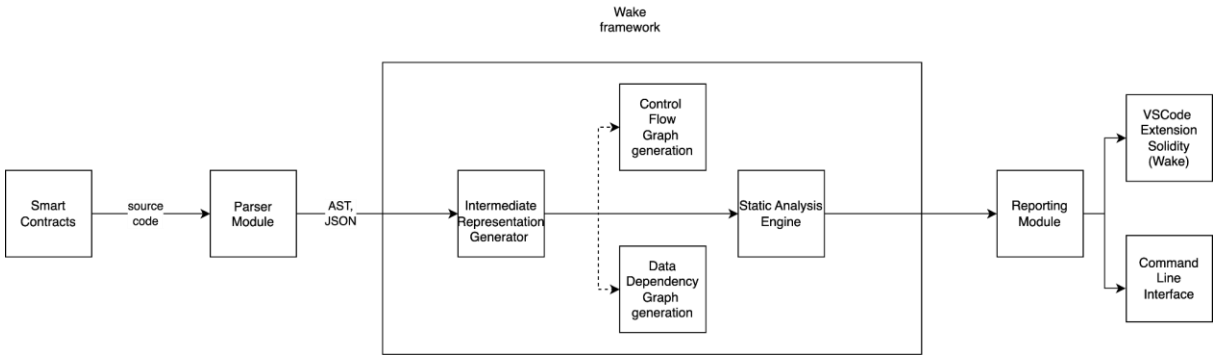

Figure 1: Wake framework overview

B. Architecture

The tool architecture consists of several modular components:

1. Parser Module: parses Solidity code into an AST for traversal. This step is done by using a Solidity compiler such as "solc"

2. Intermediate Representation Generator: converts AST into an intermediate representation (IR). The Wake framework does this transformation.

3. Static Analysis Engine: Applies predefined rules and patterns to detect potential vulnerabilities. This step is done by leveraging generated CFG and DDG by Wake framework and by using its engine.

4. Reporting Module. Generates detailed reports highlighting the issues found along with recommendations. The Wake framework provides output in two forms: in command line interface or in VSCode extension called "Solidity (Wake)".

IV. ALGORITHM FOR CFG CREATION

The Wake framework generates Control Flow Graphs (CFGs) for each function and modifier within the codebase. The framework incorporates control statements during graph construction to represent execution pathways. For conditional structures such as `if` statements, the framework creates bifurcating paths: one edge representing the execution branch when the condition evaluates to true, and a second edge representing the execution when the condition evaluates to false. The corresponding code implementation and its graphical representation are illustrated below.

The Control Flow Graph (CFG) for the compare function would be built as follows:

```
contract AContract {
  function compare(uint256 a, uint256 b) public pure returns (uint256) {
      uint256 c;
      if (a > b) {
         c = a;
      } else {
         c = b;
      }
      return c;
  }
}
```

Figure 2: Solidity code example

1. The function body is parsed to extract its statement sequence and control structures.

2. Block Construction Process:

a. *Entry Block Generation*: The algorithm creates the entry block containing the function signature and local variable declaration `uint256 c;`.

b. *Control Structure Analysis*: The if-else statement is identified as a conditional branching point.

c. *Conditional Block Formation*: The condition expression `a > b` is isolated as a discrete block with bifurcating control paths.

3. Edge Creation Mechanics:

a. The conditional block establishes two outgoing directed edges—one to the True block and one to the False block.

b. Each conditional branch block (True/False) contains exactly one outgoing edge to the Merge block, representing control flow reconvergence.

c. The algorithm detects that both branches terminate without return statements, necessitating a common merge point.

4. Path Completion:

a. The Merge block is created to represent the point where control flow from both conditional branches reconverges.

b. The return c; statement is placed in this block, representing the common exit path for the function regardless of which branch was taken.

c. This structured approach ensures that the resulting CFG accurately captures all possible execution paths through the function while maintaining the semantic relationships between code blocks.

V. ALGORITHM FOR DDG CREATION

In contrast to construction of the CFGs, one DDG is created for the whole codebase. This graph consists of nodes, which represent all occurrences of the variable in the codebase. Besides, nodes can represent intermediate calculations, such as binary operations. In-edges represent write operations and out-edges represent read operations respectively. As an

example, in this article we analyze the DDG from the previous function.
The Data Dependency Graph (DDG) for the compare function consists of five distinct nodes representing variables and operations:

- **Node 0**: Parameter variable uint256 a
- **Node 1**: Parameter variable uint256 b
- **Node 2**: Function return value (return node)
- **Node 3**: Local variable uint256 c
- **Node 4**: Binary operation expression a > b (conditional predicate)

The directed edges within the graph represent data flow dependencies between these nodes:
- Variable **b** establishes two outgoing dependencies:
  - serves as the left operand in the binary comparison operation a > b
  - provides the value assigned to variable c when the conditional evaluates to true
- Variable **b** similarly establishes two outgoing dependencies:
  - serves as the right operand in the binary comparison operation a > b
  - provides the value assigned to variable c when the conditional evaluates to false
- Variable **c** receives an incoming dependency from either variable **a** or variable **b**, contingent upon the evaluation result of the conditional expression
- The function return value (Node 2) maintains a direct incoming dependency on variable **c**

This graph structure comprehensively captures all data flow relationships within the function, illustrating value propagation and variable interactions throughout the execution paths.

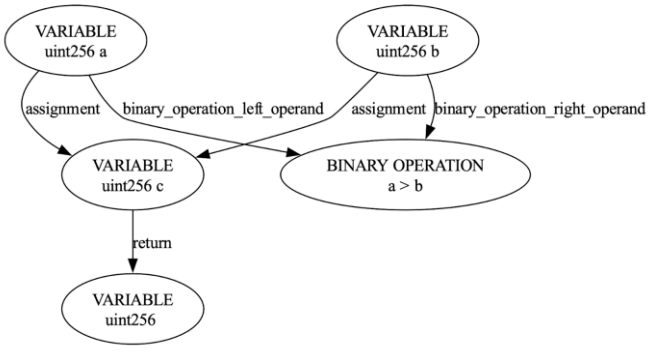

Figure 3: Data Dependency Graph example

VI. Connection between CFG and DDG

During the design and implementation of vulnerability detection algorithms, researchers typically utilize two complementary graph representations to enhance precision. The Data Dependency Graph (DDG) and Control Flow Graph (CFG) provide distinct yet interrelated perspectives on program semantics. In the DDG structure, in-edges signify read operations on variables, while out-edges signify write operations. These operations correspond directly to specific statements represented as elements in nodes within the CFG. This structural correlation between the two graph representations enables analysis by combining control flow information with data dependency relationships. As a result, static analysis detector implementations can traverse between representations—locating a specific DDG node's corresponding position within the CFG structure, and conversely, extracting relevant DDG nodes from a given CFG node when data dependency information is required for analysis.

VII. Detector for Identifying Write_After_Write Issue

The write-after-write pattern represents a critical vulnerability in smart contract development, characterized by a variable being overwritten before its previous value is utilized. This results in potential data loss and state inconsistencies within the contract execution flow.
This vulnerability is particularly significant in cross-contract interaction scenarios. When a smart contract performs a write operation and subsequently calls an external contract without validating the return value, the contract may operate on obsolete or invalid data, propagating errors or introducing security vulnerabilities.
Mitigating write-after-write patterns enhances code quality and fortifies contracts against potential exploitations stemming from unvalidated state transitions. The following algorithm is implemented to detect these vulnerability patterns:

1. **Syntactic and Semantic Analysis**: The detector parses the smart contract source code using the parser module to generate an Abstract Syntax Tree (AST), transforms it into an Intermediate Representation (IR), and constructs Control Flow Graphs (CFG) and Data Dependency Graphs (DDG) for the analyzed codebase.
2. **Variable-Centric Analysis**: For each non-storage variable node in the DDG:
2.1. Extract the complete set of read operations (represented by out-edges) and write operations (represented by in-edges) associated with the target variable from the DDG.
2.2. For each write operation W_1 (represented by an in-edge) to the variable, determine if a subsequent write operation W_2 can occur without an intervening read operation. This determination involves:
1. identifying the corresponding CFG node N_1 containing the statement for write operation W_1;
2. locating all CFG nodes $\{N\_r_1, N\_r_2, ...\}$ containing statements for read operations of the variable;
3. executing a path analysis algorithm on the CFG to determine if there exists any path P from N_1 to any node in $\{N\_r_1, N\_r_2, ...\}$ that does not traverse through any CFG node containing another write operation to the same variable; and
4. if no such path P exists, flag W_1 as an unused write operation.

3. **Vulnerability Reporting**: Present identified unused write operations to developers through the console interface or as

inline annotations within the Integrated Development Environment (IDE).

VIII. Performance Evaluation

Evaluating a detector's performance using static code analysis to find security issues is important to ensuring the identification of vulnerabilities and minimizing false positives and false negatives. This evaluation also ensures the detector can be integrated into existing development workflows, allowing for efficient and practical use in real-world scenarios. As a result, we decided to compare our developed detector in the Wake framework with the detector implemented by the state-of-the-art tool Slither. During evaluation, we performed two experiments aimed at comparing the results of both detectors in different scenarios, which are described below. Comprehensive methodological details and extended experimental results are documented in the related research publication [9]. Furthermore, the corpus of smart contracts utilized for validation testing has been archived and made available in the aforementioned reference for reproducibility purposes.

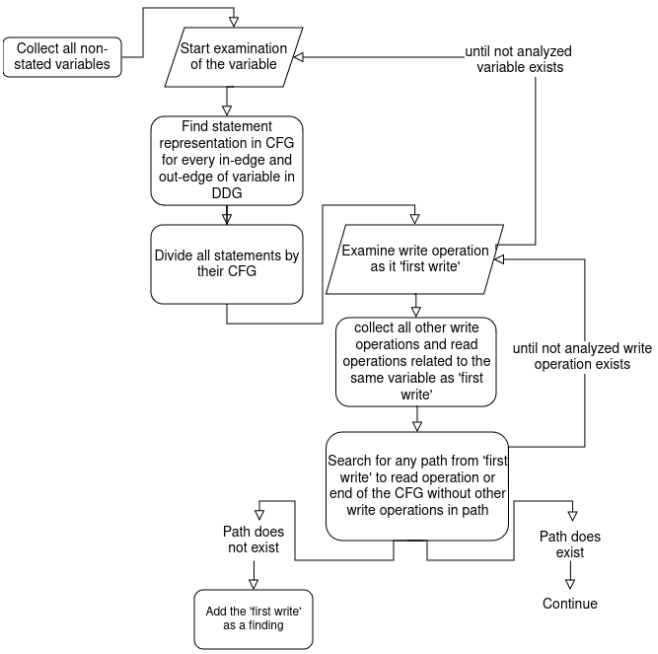

Figure 4: Algorithm of "write-after-write" detector

A. First Experiment

Thirty-four prepared smart contracts were tested during this experiment. These smart contracts are specially developed for testing the "write-after-write" detector. They contain simple cases and edge cases, which are difficult to find. The set of these smart contracts can be found in the attachment to this document. Used metrics for the evaluation:

- **Precision**: The positive predictive value measures the ratio of predicted true positives among all instances classified as positive.
- **Recall**: Also known as the true positive rate, it calculates the proportion of correctly predicted true positives out of all actual positives.
- **F1-Score**: The harmonic mean between precision and recall.

After a manual review of the contracts, it was found that there are 14 contracts that do not have the write-after-write issue and 20 contracts that are found to have this finding. The results are introduced below:

	Slither	Wake
Precision	86.36%	100%
Recall	33.33%	76.19%
F1-Score	50%	86.59%

Table 1: Precision, Recall, F1-Score comparison – First experiment

It should be mentioned that the purpose of this set of smart contracts is to test detectors on simple and edge cases.

B. Second Experiment

This investigation selected a subset of vulnerable smart contracts from the dataset by Rossini[10]. This dataset includes the results of an analysis conducted by Slither's "write-after-write" detector, and due to its size, only 15,000 entries were chosen. During experiment Slither outputs were taken from the dataset and Wake outputs were received by running detectors on smart contracts from the dataset. The processing of this dataset followed these steps:

1. The addresses to analyze were extracted.
2. The findings from Slither's "write-after-write" detector were retrieved for the extracted address.
3. The findings from Wake's "write-after-write" detector were obtained by analyzing the same set of addresses.
4. As Slither detected fewer findings than Wake, it was decided to manually analyze findings that were not found by the Wake framework that had been identified by Slither.
5. Due to time constraints, it was impossible to analyze all smart contracts manually. Therefore, a random set of 50 smart contracts has been chosen to be evaluated.

Besides, it should be mentioned that the Wake detector "write-after-write" can not scan smart contracts if at least one of the following restrictions is in place:

- Smart contract compiled by the Solidity compiler with version below 0.6.2;
- A smart contract uses absolute paths. The reason was Wake could not download these smart contracts.

As a result, only 9388 out of 15.000 smart contracts have been analyzed by implementing the "write-after-write" detector in the Wake framework.

There are 63 contracts that Slither flagged Wake did not flag. The list of these contracts can be found in the appendix. These contracts have been analyzed, and the results of the analysis of Slither findings can be seen below in the table:

Metrics	True Positive	False Positive	Precision
Slither	10%	53%	15.9%

Table 2: 63 flagged contracts by Slither manually verified

These results show that most of the findings, which have been detected by Slither and have not been detected by Wake, are False Positives.

The next table displays the number of findings identified by Wake and Slither. It can be seen that the number found by the realization of the detector using the Wake framework is larger than the number found by Slither.

This table with results shows that the implemented "write-after-write" detector using the Wake framework has more precise results than Slither's implementation of the detector.

	Slither	Wake
Precision	86.36%	100%
Recall	50%	94.74%
F1-Score	63.33%	97.29%

Table 3: Randomly chosen 50 contracts from the analyzed dataset

C. Summary

Due to time constraints, it was impossible to evaluate the results using metrics that typically involve assessing a number of false negatives. Consequently, evaluating the performance of the detectors is not straightforward. However, based on the analysis of the results presented, the following conclusions can be drawn:

- Based on results from Table 1 and Table 3, the Wake detector demonstrates greater precision than the Slither detector.
- Based on results from Table 2, the Wake detector identifies more security issues than the Slither detector does.

IX. CONCLUSION AND FUTURE WORK

This research demonstrates how the Wake framework's architecture enables the development of static analysis detectors with enhanced precision through its utilization of specific condition-based analyses rather than relying on heuristic approaches. Furthermore, the study empirically establishes how the integrated application of Data Dependency Graphs (DDG) in conjunction with Control Flow Graphs (CFG) can significantly enhance detection metrics, including precision, recall, and F1-score values.

Future research directions include the development of additional detection algorithms that leverage DDG-based analysis methodologies. Additionally, we intend to enhance existing detector implementations within the Wake framework by integrating DDG analysis techniques, thereby improving detection accuracy and reducing false positive rates. This integration will enable more sophisticated program analysis capabilities that can identify complex vulnerability patterns through combined control and data flow analysis.

REFERENCES

[1] Z. Pratap, "Reentrancy Attacks and The DAO Hack," Chainlink Blog, 2022. [Online]. Available: https://blog.chain.link/reentrancy-attacks-and-the-dao-hack/ [Accessed: Nov. 17, 2024].

[2] L. Luu, D. H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," in Proc. 2016 ACM SIGSAC Conf. Comput. Commun. Security, pp. 254–269, 2016.

[3] N. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts," in Int. Conf. Principles Security Trust, pp. 164–186, 2017.

[4] S. Grossman, G. Gurguc, M. Liu, M. Naiye, and Q. Zhang, "Offline Analysis of Blockchain-based Smart Contracts," arXiv:1808.00624, 2018.

[5] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in IEEE/ACM 2nd Int. Workshop Emerging Trends Software Eng. Blockchain (WETSEB), pp. 8–15, 2019.

[6] "Wake Framework Documentation," Ackee Blockchain. [Online]. Available: https://ackee.xyz/wake/docs/latest/ [Accessed: Jan. 12, 2025].

[7] M. Převrátil, "Extension of tool Woke," M.S. thesis, Dept. Inf. Security, Czech Tech. Univ. Prague, Prague, Czech Republic, 2022.

[8] J. Kuběna, "Woke tool extension," M.S. thesis, Dept. Inf. Security, Czech Tech. Univ. Prague, Prague, Czech Republic, 2023.

[9] [9] D. Khimchenko, "Ethereum Vulnerability Detectors" B.S. thesis, Czech Tech. Univ. Prague, Prague, Czech Republic, 2024.

[10] "Slither Audited Smart Contracts," Rossini Dataset. [Online].Available: https://huggingface.co/datasets/mwritescode/slither-audited-smart-contracts [Accessed: Nov. 16, 2024].