FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Ethereum vulnerability detectors
Student:	Dmytro Khimchenko
Supervisor:	Ing. Josef Gattermayer, Ph.D.
Study program:	Informatics
Branch / specialization:	Information Security 2021
Department:	Department of Information Security
Validity:	until the end of summer semester 2024/2025

Instructions

In DeFi and other areas where the Ethereum blockchain is used, there are still a number of attacks. One way to prevent these attacks is by using static code analyzers to identify vulnerabilities within, for example, the IDE or CI/CD process.

Wake [https://github.com/Ackee-Blockchain/wake] is an open source tool that facilitates the analysis and development of Solidity contracts. One of its main functionalities is a vulnerability and bug detection module using static code analysis. The module provides an interface that can be used to extend the Wake tool with additional detectors. Detected problems are presented to the user in the form of text output in the terminal or in the Visual Studio Code development environment.

Instructions:
- Analyze existing Wake detectors, including the IR data model used in the detectors.
- Evaluate the performance of the existing detectors on a set of contracts provided by the supervisor.
- Implement new bug and vulnerability detectors in consultation with the supervisor.
- Test the implemented detectors on the set of contracts.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 23 January 2024 in Prague.

Bachelor's thesis

ETHEREUM VULNERABILITY DETECTORS

Dmytro Khimchenko

Faculty of Information Technology
Department of Information Security
Supervisor: Ing. Josef Gattermayer, Ph.D.
May 14, 2024

Citation of this thesis: Khimchenko Dmytro. *Ethereum vulnerability detectors.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Figures

List of Tables

I want to thank Ing. Michal Prevratil for helping me dive into the world of static analysis and Ing. Josef Gettermayer, Ph.D. for mentoring me and helping with structuring my ideas. Also, I would like to thank my family for their everyday support.

Declaration

In Prague on May 14, 2024

Abstract

Nowadays, there is a tremendous popularity of technology called blockchain. This technology has many usage cases, but one of them is allowing people to develop applications and deploy them to the new environment, called blockchain, and benefit from its features like code transparency and a decentralized way of storing the code. Despite the advantages that blockchain offers, many issues can occur because of insecure logic in the code. The thesis aims to improve the security of on-chain deployed programs by scanning them for vulnerabilities using a static analysis tool provided by the "Wake" framework without interacting with an application.

Keywords blockchain, ethereum, smart contract, solidity, static analysis, vulnerability

Abstrakt

V současné době je nesmírně populární technologie zvaná blockchain. Tato technologie má mnoho případů využití, ale jedním z nich je umožnit lidem vyvíjet aplikace a nasadit je do nového prostředí, kterému se říká blockchain, a využívat jeho vlastností, jako je transparentnost kódu a decentralizovaný způsob ukládání kodu. Navzdory výhodám, které blockchain nabízí, může dojít k problémům kvůli nezabezpečené logice applikace. Cílem této bakalářské práce je zlepšit bezpečnost programů nasazených na blockchain tím, že ji bez jakékoliv interakce s aplikací ověřit na zranitelnosti pomocí nástroje statické analýzy, který poskytuje framework “Wake”.

Klíčová slova blockchain, ethereum, smart contract, solidity, statická analýza kódu, zranitelnost

List of abreviations

EVM	Ethereum Virtual Machine
PC	Program Counter
AST	Abstract Syntax Tree
CFG	Control Flow Graph
DDG	Data Dependency Graph
EOA	Externally Owned Account
SC	Smart Contract
IR	Intermediate Representation

Introduction

The word 'cryptocurrency' has become so popular [1] that it is impossible to ignore anymore. However, we should research what stands by this word and what opportunities it gives us. More and more scientists invest their energy and time into exploring the possibilities of blockchain [2], the technology that allows cryptocurrency to exist. Blockchain allows people to use their financial assets without any intermediate entity being involved and without any censorship. In my opinion, the blockchain and its primary usage, cryptocurrency, will soon be part of our everyday lives. My goal is to make the new era of decentralized finance even safer than centralized finance, regulated by a centralized authority. However, nowadays, the blockchain community experiences a lack of security, and several hacks in the blockchain environment arise [3]. To go mainstream with the new era of decentralized finance, we need, as the community, to make it secure in the first place.

The first chapter of the thesis introduces the world of blockchain, its principal terms, and the technology in use. In addition, it provides information about the programming language Solidity, which is used to develop on-chain applications for the Ethereum blockchain.

The second chapter describes the static code analysis. After that, it describes the open-source Wake framework, which has the capability to provide everyone the opportunity to design and implement a vulnerability detector.

The third chapter evaluates the performance of detectors described in the previous chapter on a defined set of smart contracts and real-world smart contracts on the main chains.

The fourth chapter introduces the new detector's development process using tools provided by the Wake framework. Moreover, it describes challenges that have been met during the implementation process.

The fifth chapter evaluates the performance of the implemented detector and compares it with existing solutions.

Objectives

The main objective of this work is to analyze possible solutions to prevent breaches and hacks on the blockchain by using static code analysis provided by the Wake framework to identify vulnerable spots of the program during the developing process. The Wake framework is an open-sourced tool for developing and auditing applications for blockchain in Solidity [4]. In addition, there is an interface that allows everyone to design and develop a vulnerability detector using static analysis. The goals of this thesis are:

- analyze the design of already implemented detectors;
- evaluate them on the provided set of the contracts;
- implement the own detector;
- evaluate its performance and compare with the same detector of state-of-the-art static analysis tool.

Chapter 1

Theoretical background

This chapter provides basic theoretical information about blockchain technology, the Ethereum project, which is fully built on this, and describes the main technological novelty brought by this project. The first chapter describes blockchain technology and its main principles. After that, the second chapter introduces Ethereum and its capabilities. The third chapter presents the language for writing applications, which will be further deployed on the Ethereum blockchain.

1.1 Blockchain

Blockchain is the technology that makes possible the realization of transparent and append-only storage updated and shared across many computers in the decentralized network. [5]

- **Block** refers to data and state, which are being stored in sequential groups known as blocks
- **Chain** means that cryptography links each block to its parent block. This creates a chain of blocks. The data in a block cannot be changed without changing all blocks he follows, which would require the entire network to validate these blocks again.

The blockchain is implemented as a mechanism for creating consensus between decentralized entities that do not need to trust each other. Created consensus means that a general agreement has been reached. The consensus mechanism is the deterministic approach to making a decision on which data might be added to the storage and which are not allowed. Every participating entity needs to follow this mechanism to exist in the network.

1.1.1 Consensus mechanisms

Numerous methodologies and strategies are available for constructing this mechanism, reflecting the various range of consensus models favored by specific blockchain networks and their operational requirements. [6].

1.1.1.1 Proof of Work

Proof-of-work (PoW) is an algorithm that sets the difficulty level and rules governing the activities of miners on PoW-based blockchains. Mining represents the 'work' needed to validate and append legitimate blocks to the blockchain. This process is crucial as it directly affects the blockchain's length, thereby helping the network determine the most accurate progression of the chain. As miners continue to solve complex mathematical problems, thereby adding more blocks, the blockchain grows longer, which improves the network's confidence in the current state of the ledger. This chain lengthening also increases security by making altering any information in previous blocks increasingly difficult. This consensus mechanism is used in the most popular blockchain implementation called Bitcoin [7].

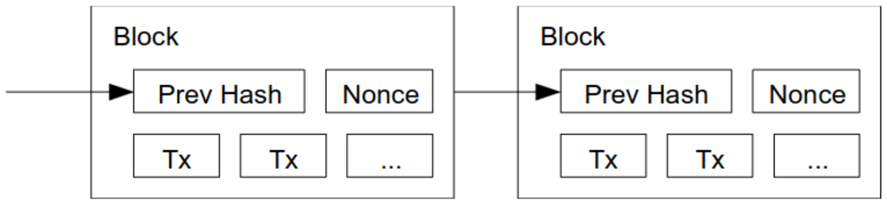


Figure 1.1 Proof of work in Bitcoin [8]

1.1.1.2 Proof of Stake

Proof of Stake (PoS) is an essential consensus algorithm that defines the rules and mechanisms for participants in PoS-based blockchains. In PoS, the process of 'staking'—rather than mining—serves as the mechanism through which validators are chosen to confirm transactions and create new blocks. Validators are selected based on the amount of cryptocurrency they hold and are willing to 'stake' as collateral. This process is critical as it helps ensure the security and accuracy of the blockchain by encouraging validators to act honestly to avoid losing their stakes. As more blocks are validated and added to the chain, the blockchain becomes longer and more robust, thereby upgrading the network's trust in the current ledger. This method increases the efficiency of the validation process and reduces the energy consumption compared to proof-of-work systems, making it a more environmentally friendly alternative.

The most used blockchain that uses the proof-of-stake mechanism is Ethereum [9].

1.2 Ethereum

Ethereum, officially launched in 2015, became an evolution in blockchain technology by allowing developers to deploy decentralized applications (dApps[1]) directly onto its platform. This innovation also uses the blockchain's inherent qualities of transparency, decentralization, and immutability, providing a robust environment for applications that benefit from these features.[10] Besides, Ethereum introduced the concept of smart contracts, self-executing contracts with the terms directly written into code, which automate and execute agreements without intermediaries. This capability has broadened the scope of blockchain, affecting industries ranging from finance to supply chain management. [11] [12].

1.2.1 Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) is the engine that powers the Ethereum network. [13] It acts like a global supercomputer, running the code for smart contracts and keeping the network secure and functioning. When developers create smart contracts using programming languages like Solidity [14], these contracts are compiled into bytecode, which is a low-level, machine-understandable language. This bytecode is what the EVM reads and executes.

When a user or another contract on the Ethereum network wants to interact with a smart contract, they send a transaction. This transaction specifies the function they want to execute and any necessary data the function needs. Once this transaction is confirmed and included in a blockchain block, the EVM activates to execute the contract's code as instructed by the transaction. The EVM ensures that the contract follows the rules of Ethereum and that everything operates seamlessly without allowing any party to cheat the system. [15]

It is necessary to mention that there are two types of accounts in the Ethereum environment:

- **Externally Owned Account (EOA)**. It is an account that is owned by any external entity. Most commonly, it is referred to as a user account. [16]
- **Smart Contract Account**. It is an account that is owned by a smart contract and often controlled by an EOA that interacts with the deployed smart contract

[1]dApp - application, that uses blockchain as its storage on the backend.

1.2.1.1 Gas

To ensure security, every operation in a smart contract requires a certain amount of "gas," which is paid in Ethereum's currency, Ether. [17] In the Ethereum ecosystem, "gas" refers to the unit that measures the computational effort required to execute specific operations on the Ethereum network. Key aspects of the gas role in the Ethereum environment:

- **Transaction Execution and Limitation**. Every Ethereum transaction, whether a simple transfer of Ether or a complex smart contract interaction, requires computational resources. Gas measures how much work is necessary to perform a transaction or execute a smart contract. Each operation in the Ethereum Virtual Machine (EVM) has a fixed gas cost.

- **Anti-Spam Mechanism**. Gas serves as a constraint against spamming the network. Since executing transactions and running smart contracts require gas, which costs Ether, malicious actors are discouraged from spamming the network because it would be financially costly.

This gas system prevents people from running faulty or malicious code on the network that could slow it down. Each operation has a gas cost, and users set a gas limit when they send a transaction to indicate the maximum they're willing to spend. If the gas runs out before the operation is completed, the transaction is reverted, but the gas spent is not refunded, protecting the network from abuse.

1.2.1.2 Memory

The Ethereum Virtual Machine (EVM) uses several types of memory storage to facilitate and optimize the execution of smart contracts. [18]

- **Storage**. This is the EVM's most permanent form of memory and is part of the Ethereum blockchain's state. Storage is used to hold the persistent state of a smart contract, including variables declared by the contract. It is relatively expensive because it is written directly to the blockchain, making changes costly and permanent. Access to storage is slow compared to other types of memory because changes need to be propagated and confirmed across all nodes in the network.

- **Memory**. This volatile type of memory is wiped clean after every transaction call. It is used to hold temporary values.

- **Stack**. The stack is used for managing internal computations. It is a low-level type of memory with a LIFO (last in, first out) structure, where data can be pushed onto or popped off the stack. The stack has a maximum size, and it is necessary to manage its use carefully to prevent overflow errors. The stack is not directly accessible by the contract's high-level code but is manipulated through the EVM's set of opcodes.

- **Calldata**. This read-only memory area contains the input data of a call to a function in a smart contract. Calldata is immutable and persists for the duration of a transaction call.

1.2.1.3 Architecture

The EVM operates in a completely sandboxed environment, ensuring that code execution within the EVM cannot affect the host machine or the main network. Each smart contract runs in its own isolated virtual space, preventing it from interacting directly with the host's system files or processes. In addition, the EVM is a stack-based machine, meaning it performs most of its computations using a stack data structure. It can push data to the stack, perform operations, and pop the results off. The stack has a limited size, which helps prevent excessive resource consumption.

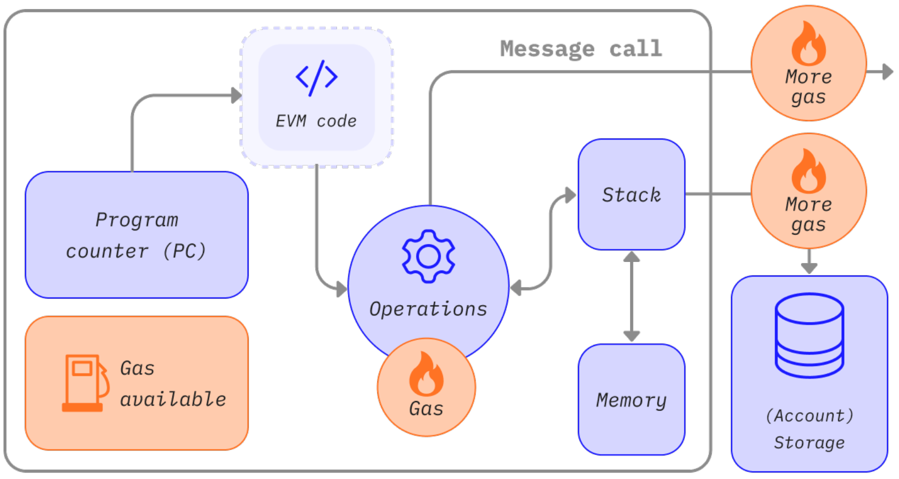


Figure 1.2 EVM architecture overview [19]

A few essential points from this scheme must be described for clarifying information:

- **Program Counter (PC)**. The program counter in the EVM tracks the current instruction that it is executing. It moves sequentially but can be altered by loop and branching instructions, similar to traditional CPUs [20].
- **Gas**. The gas icon in this scheme indicates where it is used. For example, executing any instruction costs gas, and calling other contracts or accessing the storage of the blockchain is even more gas-expensive.
- **EVM code**. This area is immutable, meaning it cannot be changed during execution. It typically holds the bytecode of the smart contract that is being executed.

- **Account Storage**. Represented as a database icon, this is part of the persistent state of the blockchain where the data of smart contracts (such as balances and states) are stored permanently.

1.2.2 Smart Contracts (SCs)

Smart contracts are self-executing applications that run on a blockchain, which makes them distributed and decentralized. [21] Smart contracts automate the execution of a code so that all participants can be immediately sure of the outcome without any intermediary's involvement or time loss. The most important features of a smart contract, which differ it from a typical application, are described below:

- **Automatic execution**. Smart contracts operate automatically without central authority or any other external entity. Once deployed on the blockchain, they automatically execute in response to predefined conditions or triggers.
- **Transparency**. The decentralized nature of blockchain ensures that all transactions and their outcomes are transparent and immutable. As a result, everyone can see every call to a smart contract.
- **Security**. Smart contracts use blockchain technology at their core, providing a high-security level. Thanks to cryptographic processes and decentralized execution, smart contracts resist hacking and unauthorized changes.

1.2.2.1 Upgradeability of Smart Contracts

The immutability of smart contracts is an essential characteristic of blockchain technology. This implies that once a smart contract is deployed on the blockchain, its code cannot be altered; it is permanently recorded on the blockchain ledger. This permanent nature ensures that the behavior of the contract cannot be changed once it is in operation, which is critical for maintaining trust and security within the Ethereum ecosystem.

Because of the immutability of smart contracts after deployment them on the blockchain, several challenges must be addressed by developers:

- **Error Correction**. Correcting these errors is not straightforward if bugs or vulnerabilities are discovered in the smart contract code after deployment.
- **Upgrading Mechanisms**. To address the issue of immutability and allow for improvements or bug fixes, developers often have to deploy a new version of the contract. This involves creating an entirely new contract and migrating all the existing data and digital assets from the old contract to the new one, which can be complex, risky, and costly. It also involves

additional steps to ensure that users interact with the updated version of the contract.

Therefore, it is essential to prevent bugs and vulnerabilities in the process of developing the smart contract.

1.2.3 Solidity

Solidity [22] is a statically typed object-oriented programming language. Its most popular use is for writing smart contracts that run on an Ethereum blockchain. [23]. This language was developed to be easy to use and with restrictions to contain as few security issues as possible. For instance, no pointers or simple data structures like queues or sets exist.

1.2.3.1 Elements of the language

For the purpose of this thesis, it is essential to understand what entities can be met during static analysis, which are essential for the particular analysis, and which are unnecessary to analyze and can be ignored.

- **Contract**. This entity is a declaration of the smart contract and is the most essential object in the Solidity programming language. It can have external and internal functions, also called smart contract methods. Besides, smart contracts can be abstract; in this case, not all their methods need to be defined. This type of smart contract is used for inheritance, and child contracts take on the burden of implementing abstract methods.
- **Modifier**. A modifier in Solidity is a special construct similar to a decorator in Python. It is used to change the behavior of functions in smart contracts by wrapping additional logic around them. Unlike functions, modifiers cannot be called directly as standalone entities. Instead, they are declared with the modifier keyword and are invoked by appending them to the function declaration. This allows them to execute logic before or after the function they modify, depending on how the modifier is structured.
- **Variable**. It represents a named object, which can store different values depending on the variable type.
- **Event**. This special Solidity entity is used for logging events on the blockchain, where smart contracts run. It is declared in smart contract objects and can be emitted after the execution of a specific branch of code occurs.
- **Error**. It represents custom types of errors defined by developers in the scope of one smart contract.

- **Structure**. A 'structure' entity is a definition of a new type of variable, which is an association of the previously defined types.
- **Enumeration**. It is a custom type of a variable that can be transformed into an unsigned integer.
- **Interface**. It is a special type of smart contract that can have function declarations but cannot implement them.
- **Library**. It is a custom smart contract deployed to the blockchain once and has functions that other contracts can use and execute in their environment.

1.2.3.2 Types of variables

As mentioned above, the primary purpose of the Solidity programming language is to serve as an instrument for writing smart contracts. Therefore, a few particular types that make sense only in the smart contract environment are also introduced in the list below. Value types are introduced at the beginning of the list, and reference types are introduced at the end.

- **uint**,**int**,**float**,**bool**. These are the simplest variable types, which have also been inherent in different languages and actively used in Solidity
- **bytes**. Data type that represents bytes. This data type can concatenate the two values by converting them previously to bytes data type.
- **address**. Special for the Solidity variable type, which represents the address in the Ethereum network. This variable type has special attributes and methods that can be used in the smart contract.
  - *call*, *staticcall*, *delegatecall*. These are different methods for invoking functions at a specified address. Calling an address typically means triggering a specific function on that address, particularly if the address corresponds to a smart contract (SC).
  - *transfer*, *send*. These methods are available if the address is declared as `address payable`. They send ether to a specified address and differ in processing the error.
  - *.balance*. It is an attribute used to find out the amount of ether the address owns.

Another group of variables in Solidity are reference-type variables, distinct from value-type variables. Unlike value types that store data directly, reference types store references to the locations in memory where the actual data is held. The Ethereum Virtual Machine (EVM) processes different memory types such as *storage*, *memory*, and *calldata* in distinct ways. The example of reference-type variables are:

- **string**. It is a dynamic array that stores a set of characters consisting of numbers, special characters, spaces, and alphabets.
- **mapping**. This type maps keys to values, which are hashed into a 32-byte hash. This hash determines the storage slot in which the value is placed. Unlike arrays or strings, mappings are only stored in *storage* and not available in *memory* or *calldata*.

1.2.3.3 Solidity functions

There are several types of functions in smart contracts. [24]

- **public** and **external** functions. These functions can be called by anyone in the environment of the Ethereum blockchain. An example of this function is displayed below:

```
function giveMagicNumber() public returns(uint256) {
    return 42;
}
```

- **private** and **internal** functions. These functions can be called only internally inside of the contracts or by a child contract, which inherits the internal function from the parent contract. An example of the usage of this function can be seen below:

```
function _internalMagicNumber() private returns(uint256){
    return 1337;
}

function giveMagicNumber() public returns(uint256) {
    return _internalMagicNumber();
}
```

1.2.3.4 Usage of smart contract functions

Smart contracts provide opportunities for users, and other smart contracts call their external functions. This is one of the most critical features in the Ethereum ecosystem, as it lets smart contracts use each other functions and benefit from it. A smart contract can call external functions of another contract using different types of calls.

- `.call()`: it is a common call from one contract to another with or without specification of the function to call and parameters that the calling smart contract wants to pass to the function. This call can change the state of the blockchain and be executed in the called contract environment.

- `.staticcall()`: it is a similar call as `.call()` except that it reverts if a state change of the blockchain happens. Using this type of call, sending ether to another smart contract is impossible.
- `.delegatecall()`: it is a call that executes the code of another contract, which is called, but in the environment of the calling contract.

1.2.3.5 Compilation and Deploying process

A smart contract code is typically compiled by an official solc [25] compiler. After compilation, 'solc' generates **bytecode** and Application Binary Interface **ABI**.

- **bytecode**. It is a sequence of bytes that EVM can read. Besides, there are two main parts of the bytecode (can be seen in the Figure 1.4):
  - *Contract creation bytecode.* It is bytecode, which is executed during the initialization of the contract
  - *Contract runtime bytecode.* It is bytecode, which is executed when any interaction with the deployed smart contract is happened.
- **ABI**. JSON-formatted description of how to interact with a smart contract, including its methods and structures.

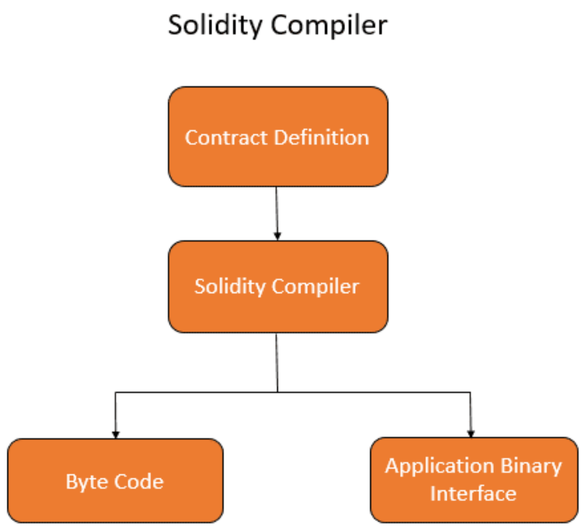


Figure 1.3 Outputs from compilation process by solc [26]

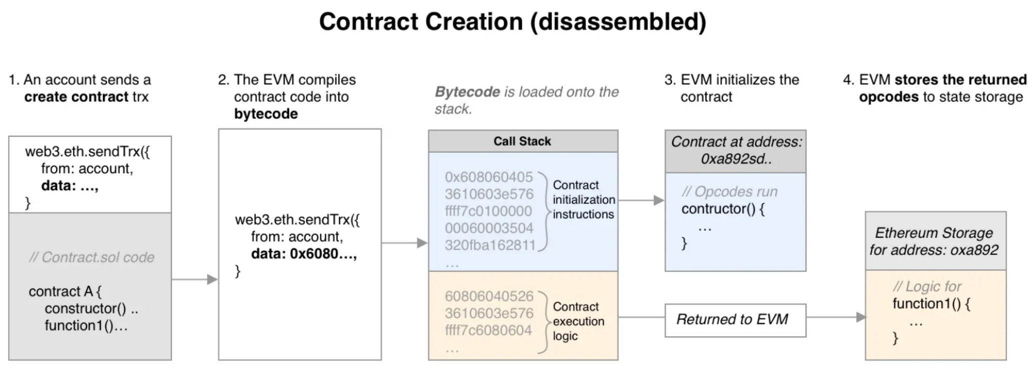

Contract Creation (disassembled)
1. An account sends a create contract trx
web3.eth.sendTrx({ from: account, data: ..., }
// Contract.sol code
contract A { constructor() .. function1()...
2. The EVM compiles contract code into bytecode
web3.eth.sendTrx({ from: account, data: 0x6080..., }
Bytecode is loaded onto the stack.
Call Stack
0x608060405 3610603e576 ffff7c0100000 00060003504 320fba162811 ...
Contract initialization instructions
60806040526 3610603e576 ffff7c6080604 ...
Contract execution logic
3. EVM initializes the contract
Contract at address: 0xa892sd..
// Opcodes run contructor() { ... }
Returned to EVM
4. EVM stores the returned opcodes to state storage
Ethereum Storage for address: oxa892
// Logic for function1() { ... }

Figure 1.4 Deploying the bytecode to the Ethereum blockchain [27]

Chapter 2
The Wake framework

This chapter provides an overview of Wake's static analysis process. Firstly, it describes the static code analysis procedure and the main challenges. Secondly, it introduces how Wake does static analysis and what tools are already built-in in Wake to enable static analyzer developers to use them. Finally, it presents examples of detectors that have already been implemented. A detector is a tool that detects an insecure or unwanted program behavior.

2.1 Static analysis

Static code analysis [28] is a process of reasoning about a code without running it. The programs that reason about another program are called program analyzers. Static analysis has found various applications during the process of the development of programs, such as:

- finding the errors, bugs, and vulnerabilities in code without any interaction with it;
- optimization of the code by a compiler;
- coding support, e.g., providing a recommendation for refactoring;

2.1.1 Process of static analysis

Taking into consideration that the process of every program analyzer can differ, the static code analysis can be universally divided into the following steps:

1. **Code Parsing**. A program analyzer parses the application's source code into a structured format called abstract syntax tree(AST), which represents the syntactic structure of the code. An example of an Abstract Syntax tree is shown in Figure 1.1.

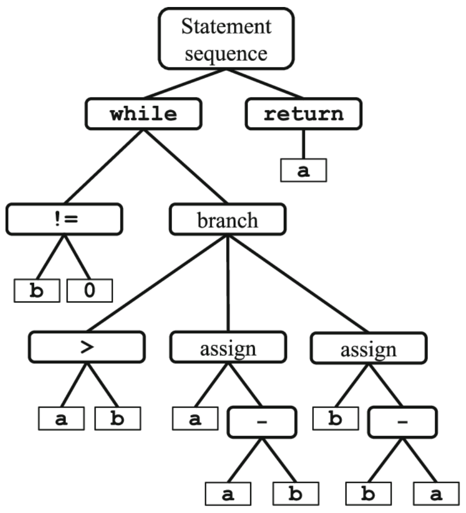


Figure 2.1 Example of Abstract Syntax Tree

2. **Intermediate Representation (IR)**. In this step, the program analyzer builds an intermediate representation of a program on top of AST. IR might be defined strictly by the compiler or the program analyzer. The main reason to use IR rather than AST is IR is more structured and can be proceeded by a program analyzer more efficiently.

3. **Semantic analysis**. Once the IR is built, the program analyzer can examine the program using the following methods:

   - **Type checking**. The tool looks into operations between variables and checks if an operation can be performed on compatible types.
   - **Data Flow analysis**. This involves analyzing the paths through which data flows through the code and helps identify potential issues. This kind of analysis can be useful when combined with control flow analysis.
   - **Control Flow analysis**. This involves inspecting the paths along which the program's control can flow. This analysis helps identify logical errors, such as infinite loops, unreachable code, and incorrect branch handling.

4. **Rule-Based Analysis**. In this step, the tool tries to detect issues in the code by searching for predefined patterns. These rules can cover code style or potential bugs.

5. **Metric Calculation**. The tool calculates various metrics, such as cyclomatic complexity, which measures the complexity of a program based on the number of independent paths through the code. High complexity can indicate code that is difficult to test and maintain.

6. **Security Analysis**. The program analyzer scans the code for patterns that match common security vulnerabilities.
7. **Reporting**. In this final step, the tool reports findings that were indicated previously.

The order of steps is not strictly defined and can differ from tool to tool. Besides, steps 2-6 are optional and may not be provided by some program analyzers.

2.1.2 Challenges of static analysis

Even though correctly developed program analyzers can improve the security of written code, they mainly face the following comprehensive problems:

- **Amount of False Positives**. Most program analyzers use a heuristic rather than a systematic approach when examining a program. Heuristics are alternatives that aim to provide quick answers without analyzing every possible scenario or state in a program. These rules rely on generalizations that might not consider all specific details and context. However, if a static analyzer uses a more systematic approach, it can reduce the number of mistakes.
- **Performance cost**. Due to the amount of work a static analyzer has to accomplish, the analysis process may be time and resource-consuming. This can slow development, especially if the analysis is integrated into the build process.

2.2 Wake internal model

Wake is the open-source Python-based Solidity development and testing framework that provides functionality for:

- Developing smart contracts;
  - coding support;
  - deployment on local chain or mainnet;
- Auditing smart contracts using:
  - fuzzing;
  - static code analysis;
- Manager of solidity compilers;

We will mainly focus on capabilities for static analysis from Wake.

2.2.1 Preparation of IR

Wake utilizes the solc-bin[1] compiler to compile all smart contracts within the codebase into bytecode, readable by the Ethereum Virtual Machine (EVM) and to generate an Abstract Syntax Tree (AST) for each contract. Subsequently, Wake parses and unifies these ASTs, enriching them with additional attributes and functions to facilitate future analysis and converting them into Python objects represented by Intermediate Representation (IR).

Additionally, during post-processing of the entities, Wake serializes Intermediate Representation (IR) objects and uses Python's pickle module[2] to save the representations in the file located at *.wake/build/build.bin*. Additionally, Wake generates a local cryptographic key to sign the hash of the dump file containing all generated objects. This signature is stored in *.wake/build/build.bin.sig* and is verified each time Wake accesses this file. This security mechanism prevents the risks of importing maliciously crafted Python serialization files.

2.2.2 Built-in tools for static analysis

The Wake framework provides a platform for designing and developing vulnerability and bug detectors, accessible to everyone through an open-source approach. It is equipped with advanced built-in functionalities such as the generation of Control Flow[3] and Data Flow[4] graphs. These tools enable a more precise code base analysis, significantly reducing the amount of False Positives[5], which is challengeable for any program analyzer.

2.2.3 Interface for developing detectors

The Wake framework provides an easy way of implementing additional detectors. The new detector can be initialized by manually creating the file in a Wake project directory as *"detectors/name-of-the-detector.py"* file. The structure of a simple detector is introduced by the code below:

[1]`https://github.com/ethereum/solc-bin/`
[2]`https://docs.python.org/3/library/pickle.html`
[3]completely open-sourced
[4]currently under development, close-sourced for now
[5]incorrect detections

```python
class MyNewDetector(Detector):
    _detections: List[DetectorResult]

    def __init__(self) -> None:
        self._detections = []

    def detect(self) -> List[DetectorResult]:
        return self._detections

    @detector.command(name="my-detector")
    def cli(self) -> None:
        pass
```

The `MyNewDetector` class inherits the `Detector` class from the library provided by Wake. The new class has to implement the following functions to be executed by the Wake static analyzer.

1. *__init__(self)* $\rightarrow$ *None* function. It is mainly a constructor for the detector, where all needed variables might be specified.

2. *detect(self)* $\rightarrow$ *List[DetectorResult]* function. This function is executed at the end of the detector work, and the main objective is to return the results of the detector's work. This function returns the list of `DetectorResult` objects found. `DetectorResult` is the single object generated by helper functions that we look at next.

3. *cli(self)* $\rightarrow$ *None* function. It is a command line interface method that uses the Click[6] library to provide a simple and structured command line interface. In this function, we define the name using which specified detector can be used.

Even though the needed parts of the detector have been introduced, special methods exist, which start from the *visit_* word. These methods are inherited from the 'Visitor' class and could tremendously help during static analysis. A simple diagram of classes in the detector may be seen in Figure 1.2.

[6] `https://click.palletsprojects.com/en/8.1.x/`

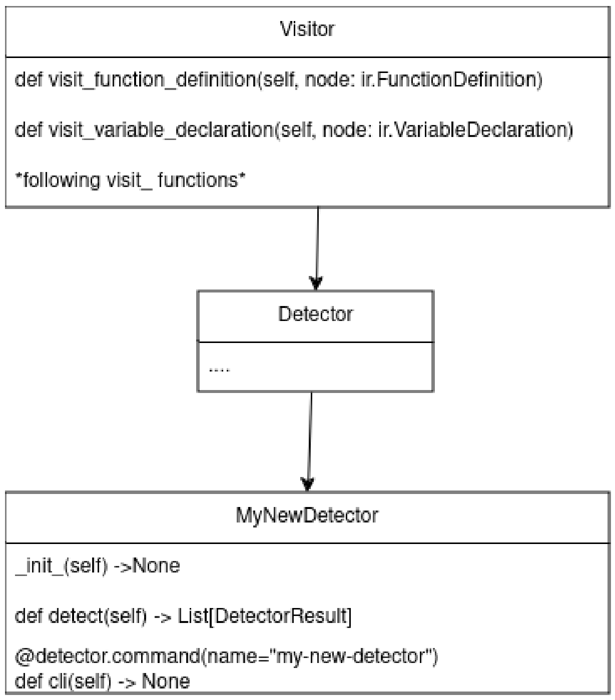

Visitor
def visit_function_definition(self, node: ir.FunctionDefinition)
def visit_variable_declaration(self, node: ir.VariableDeclaration)
following visit_ functions
Detector
....
MyNewDetector
init(self) ->None
def detect(self) -> List[DetectorResult]
@detector.command(name="my-new-detector")
def cli(self) -> None

Figure 2.2 'Detector' class inheritance

2.2.3.1 'visit_' inherited methods

The inherited `Visitor` class provides methods with a particular purpose within Wake static analysis. The `visit_` methods are dedicated to analyzing all the types of Solidity abstract syntax tree (AST) nodes. These functions are automatically called by the execution engine when running the detector. It should be noted that all AST nodes have intermediate representations (IR) nodes, which are the ones analyzed by Wake's static analysis. The specific `visit_` method visits every IR node with the same type of IR node as the parameter in this method. An example of the `visit_` method, implemented in the `MyNewDetector` class, iterates over all IR nodes of the `ir.ContractDefinition` type that represents contract definition and prints all the names of the examined smart contracts to the console.

```python
class MyNewDetector(Detector):
    _detections: List[DetectorResult]

    def __init__(self) -> None:
        self._detections = []

    def detect(self) -> List[DetectorResult]:
        return self._detections
    def visit_contract_definition(self, node: ir.ContractDefinition):
        print(node.name)

    @detector.command(name="my-new-detector")
    def cli(self) -> None:
        pass
```

2.2.4 Basics of working with IR

Wake IR is constructed based on AST generated by the solidity compiler. It is a tree representation of the source code, enriched with additional information that helps during the static analysis process. All nodes in Wake's IR model could be divided into 5s categories:

- **Declarations**. Nodes that represent declarations of variables, functions, structs, and other similar elements.
- **Statements**. Nodes that manage the execution flow (if, for, while, etc.) and nodes that represent a standalone operation ending with a semicolon (assignment, function call, etc.).
- **Expressions**. Nodes that commonly have a value (literals, identifiers, function calls, etc.).
- **Type names**. Nodes that represent a name of a type (unit, bytes, etc.) are normally used in a `VariableDeclaration` class.
- **Meta**. Nodes not belonging to the above categories are commonly used as helpers.

2.2.4.1 IR Nodes structure

The structure of IR nodes can be complex. There are a few rules that help to orientate within it.

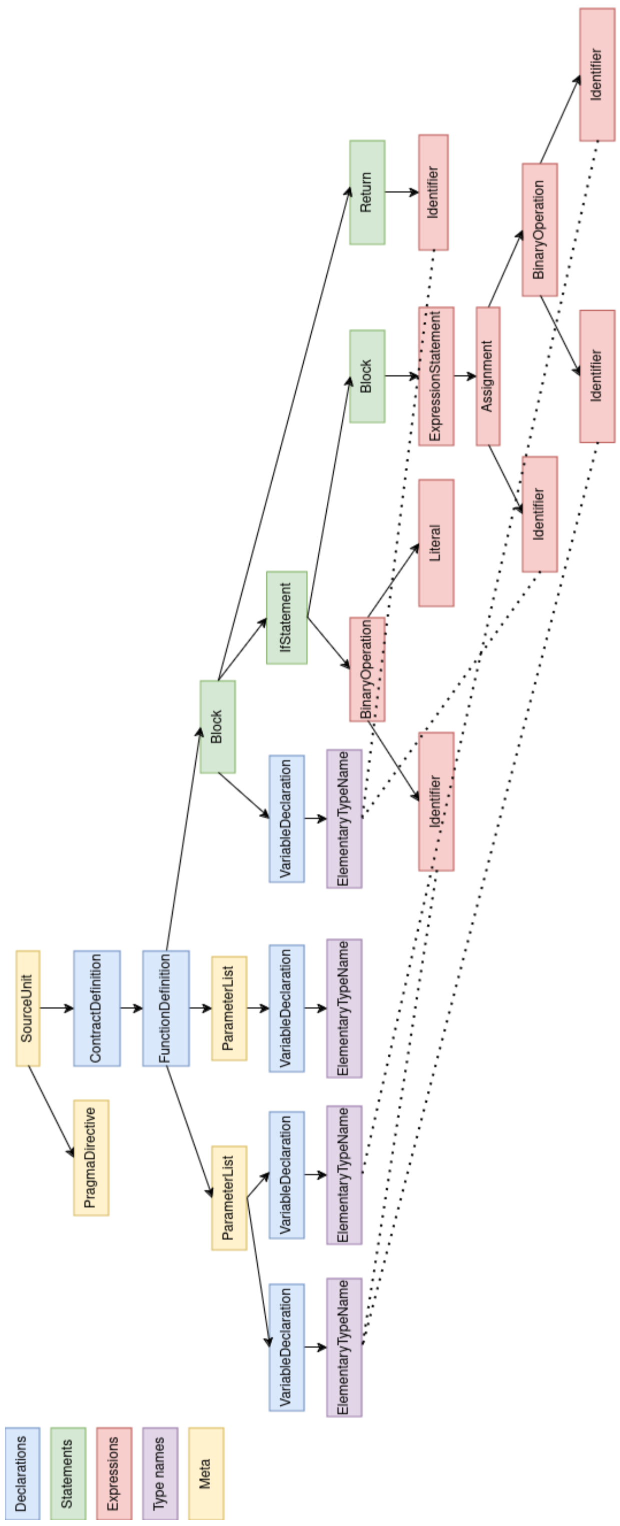
- `SourceUnit` IR class is the root node of every tree
- `FunctionDefinition` and `ModifierDefinition` classes hold statements.
- Statements might hold other statements and expressions.
- There are a few cases when expressions may be used without a parental statement:
  - in an `InheritanceSpecifier` argument list: `contract A is B(1, 2) {}`
  - in a `VariableDeclaration` initial value: `uint a = 1;`

  - in an `ArrayTypeName` fixed length value: `uint[2] a;`
- Only a few nodes might reference other nodes:
  - `Identifier`, which refers to a variable declaration (example: `owner` references the `VariableDeclaration` object);
  - `MemberAccess`, representing member access (example: `owner.balance` references the global symbol *ADDRESS_BALANCE*);
  - `UserDefinedTypeName`, as a reference to a user-defined type (example: `MyContract` in `new MyContract()`);
  - Other cases also exist but occur less frequently than those mentioned.

The following example shows the whole IR tree for the following Solidity code snippet:

```
pragma solidity ^0.8;

contract SimpleContract{
    function SimpleFunction(uint256 a, uint256 b) pure public
        returns(uint256){
        uint256 c;
        if (a < 100){
            c = a + b;
        }
        return c;
    }
}
```



■ **Figure 2.3** Example of IR tree

2.3 Built-in tools for static analysis in Wake

The structural approach for developing detectors is preferred. Because of that, developers of the Wake framework implemented the following helper tools, which will significantly assist in developing detectors.

2.3.1 Control Flow Graph

A Control Flow Graph (CFG) is a computer science data structure used primarily in program static code analysis. It represents all paths in the single graph that might be traversed through a program during its execution. The Wake framework provides. Components of the control flow graph:

- **Nodes**. Each node represents a basic block, a sequence of statements or instructions with one or more entry points and one or more exit points. A basic block is a straight-line code sequence with no branches. In the Intermediate Representation model, nodes are objects of `CfgNode` type. This object contains a few attributes, such as:
  - **id**. It is the unique identifier of the node in the single CFG.
  - **statements**. This attribute contains the list of all statements that are in this particular node.
  - **control_statement**. It is an optional attribute that identifies if any control statement is at the end of the `CfgNode`.
- **Edges**. Edges in the CFG represent the control flow from one block to another. An edge from block A to block B means that the execution can pass from the end of A block to the beginning of B block. These can be due to different reasons, such as:
  - direct jumps;
  - loops;
  - conditional branches.

This structure helps to analyze the flow and structure of program execution, making the CFG a mandatory tool for static code analysis. An example of the Control Flow graph for the code presented above can be seen below:

In addition, it is worth mentioning that every function of a smart contract has its own Control Flow Graph.

2.3.2 Data Dependency Graph

A Data Dependency Graph (DDG) is a tool used in computer science, particularly in compiler design, software engineering, and program optimization, to represent the dependencies of data elements on each other within a program.

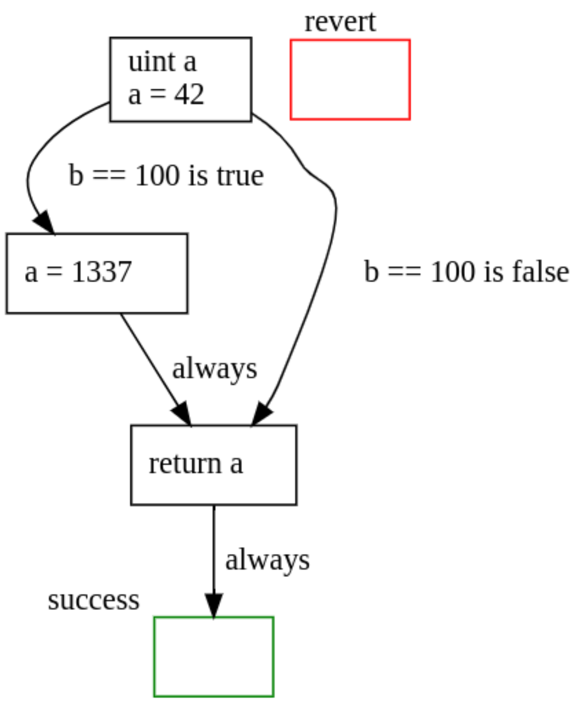


Figure 2.4 Example of Control Flow Graph

This graph illustrates how data values are related based on the program's operations, and it is used in static code analysis to optimize the program and find potential security issues. The components of a Data Dependency Graph are:

- **Nodes**. In a DDG, each node typically represents an operation or instruction within the program that produces or consumes data. For example, a variable in DDG is represented as a node.
- **Edges**. The edges between the nodes represent the data dependencies. An edge from node A to node B indicates that the operation at node B somehow depends on data produced by the operation at node A. Besides, the Wake framework has the functionality to differ between types of edges. For example, edges representing assignment operation have type `DdgEdgeKind.ASSIGNMENT`. This built-in feature enables one to distinguish between different edges and make decisions depending on the kind of edge.

DDGs aid in understanding complex program behaviors, debugging, and performing static code analysis to find potential issues like deadlocks, unnecessary serialization and issues like 'write-after-write'. The example of the part of the DDG is depicted below:

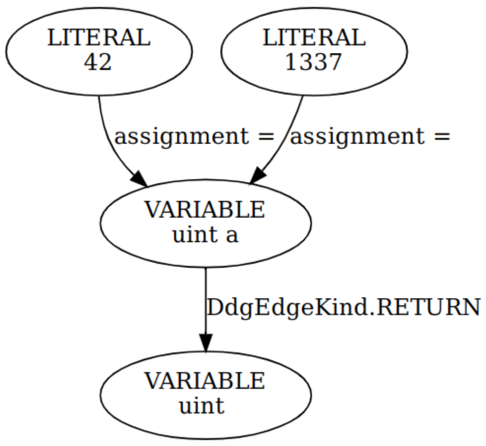


Figure 2.5 Example of Data Dependency Graph for one variable

2.4 Existing detectors

The primary motivation for developing such a structured and comprehensive IR model of the programs is the opportunity to design and implement detectors that will have the capability to discover complex bugs, errors, or vulnerabilities. This section describes the design and implementation of the 2 detectors using the Wake framework that have been specifically chosen after consultation with the thesis supervisor. These detectors have been chosen as they have the most common characteristics of the Wake realized detectors and can represent issues that can be met during the implementation process of the tool for static analysis.

2.4.1 'Call options not called' detector

The Ethereum Virtual Machine (EVM) allows smart contracts to call functions of other smart contracts on the same chain. This functionality significantly extends the capabilities of smart contracts, enabling more complex, interconnected systems within the blockchain environment. For instance, a smart contract can leverage the functionality and state of another contract to receive data from it. A smart contract can do that by making a call with appropriate data to the address where another contract is located. A smart contract that initiates a call to another smart contract can configure the following parameters:

- **amount of internal currency** of blockchain that smart contract wants to send to another contract;

- **gas limit**, which calling smart contract agrees to pay;
- **data payload**, where the called function is specified with appropriate parameters;
- **call type**, different call types are described in Chapter 1.

An example of such a call is provided below in the code:

```
function send_call_to_contract(address arbitraryContract) public {
    arbitraryContract.call{value: 1000, gas: 10000}(
        abi.encodeWithSignature("deposit(uint)", 0)
    );
}
```

In this snippet of code, the common call to another contract can be divided into the following parts:

- `{value: 1000}` - amount of internal cryptocurrency is to be sent;
- `{gas: 10000}` - amount of gas that the contract is willing to pay for this call
- `.(abi.encodeWithSignature("deposit(uint)", 0))` - data payload, where we use `abi.encodeWithSignature()` function to encode the selector of function the smart contract wants to call and specified parameters.
- `.call` - the call the smart contract is programmed to execute.

It should be mentioned that this code is unsafe; it is provided only as an example of the function call and is not recommended for use in a production environment.

Besides, it should be mentioned that in older Solidity versions before 0.7.0, it could be possible to make a call from the contract using the syntax demonstrated below:

```
contract AContract {
    function foo(address arbitraryAddress) public payable {
        arbitraryAddress.call.value(1).gas(1)("");
    }
}
```

Where parameters `value`, `gas`, and other possible ones are passed using 'function calling' syntax. Now, it is deprecated and cannot be compiled with new solidity compilers due to *TypeError*. However, the code can be refactored to the up-to-date syntax:

```
contract AContract {
    function foo(address arbitraryAddress) public payable {
        arbitraryAddress.call{value:1,gas:1}("");
    }
}
```

There is a possibility in Solidity that the function call is omitted, and instead of an actual call, the access to the member `.call` is done. The primary purpose of the 'Call options not called' detector is to search for this omitting.

2.4.1.1 Design of the detector

It is essential to design detectors compatible with older versions of the analyzing programming language. Therefore, during the design process, it is important to examine all possible older structures of the version with which the detector has to be compatible.

The example of insecure code compiled with an older Solidity version is stated below:

```
contract AContract {
    function foo(address arbitraryAddress) public payable {
        arbitraryAddress.call.value(1).gas(1);
    }
}
```

The program uses call options, but it does not call address. In the newer version of the solidity, the finding appears in this code:

```
contract AContract {
    function foo(address arbitraryAddress) public payable {
        arbitraryAddress.call{value: 1, gas: 1};
    }
}
```

There are two intermediate representation classes by iterating on which the needed information for catching the error can be extracted.

- `ir.FunctionCallOptions` - this class represents the IR node, which is responsible for call options (for example: `{value: 1, gas: 1}` in `randomAddress.call{value:1, gas:1}()`.
- `ir.MemberAccess` - this class represents the IR node responsible for accessing the object member. IR nodes of this type have the property `.referenced_declaration`, which represents the accessed attribute or function.

It is beneficial to introduce a few more entities from IR that aid in the implementation process:

- `ir.enums.GlobalSymbol.FUNCTION_GAS`, `ir.enums.GlobalSymbol.FUNCTION_VALUE` - symbols that are codified by the IR model and can only be referenced by `Identifier (wake.ir.expressions.identifier.Identifier)` nodes. In this case, the symbols are specifically ’**value**’ and ’**gas**’.
- `ir.FunctionCall` - IR node represents all operations using `()` symbols. In the particular case of implementation of this detector, this class will represent function calls.

Detector for an **older** version can be implemented in a few steps:

1. Iterate by every `ir.MemberAccess` entity;
2. if the property `.referenced_declaration` of the `ir.MemberAccess` entity is not:
   - `ir.enums.GlobalSymbol.FUNCTION_VALUE`
   - `ir.enums.GlobalSymbol.FUNCTION_GAS`

   detector continue on the next `ir.MemberAccess` entity;
3. the detector needs to parse through every valid parameter of calling options (e.g., `.valid()` or `.gas()`). It continues to iterate by parameters going upper in the IR model tree.
4. When it ends parsing through calling options, the IR node must be the type of `ir.FunctionCall`. If it is not, it indicates that call options are not called.

The code snippet that demonstrates the explained logic behind the detector is introduced below:

```
def visit_member_access(self, node: ir.MemberAccess):
    if node.referenced_declaration not in {
        it. enums.GlobalSymbol.FUNCTION_GAS,
        ir.enums.GlobalSymbol.FUNCTION_VALUE,
    }:
        return
    expr = node
    while True:
        if (
            isinstance(expr, ir.MemberAccess)
            and isinstance(expr.parent, ir.FunctionCall)
            and expr.referenced_declaration
            in {
                ir.enums.GlobalSymbol.FUNCTION_GAS,
                ir.enums.GlobalSymbol.FUNCTION_VALUE,
            }
        ):
            expr = expr.parent.parent
        elif isinstance(expr, ir.FunctionCallOptions):
            expr = expr.parent
        else:
            break

    if not isinstance(expr, ir.FunctionCall):
        # Finding
        pass
```

Detector for a new version can almost be implemented similarly but with iterations by IR nodes of type `ir.FunctionCallOptions`. The complete code of this detector can be found in the attached file.

2.4.2 'Struct mapping deletion' detector

In Solidity, mappings within a struct are not cleared when the `delete` keyword is used on the struct instance. This is due to the inherent nature of how mappings work in Solidity. Mappings are a key-value data structure used extensively due to their efficiency and flexibility. When a mapping is declared within a struct, it embeds a potentially dynamic associative array inside a more statically defined object.

The problem occurs in the following code:

```
contract A {
    struct Account {
        string name;
        mapping(uint => uint) balances;
    }

    mapping(uint => Account) accounts;

    function clearAccount(uint id) internal {
        delete accounts[id];
    }
}
```

The function `clearAccount` is intended to delete all information from the struct object 'Account' connected to the specified `key` of unsigned integer type. However, due to the dynamic nature of the mapping 'balances' structure, it cannot be deleted with a simple `delete` keyword. For example, the static attribute 'name' of the 'Account' struct can be cleared with `delete` keyword in the way that the value of the `Account.name` will be empty string (`""`).

2.4.2.1 Design of the detector

To find this issue, we need to iterate through all calls of `delete` and find if the second operand has in itself a struct with mapping. IR nodes that can be used during the implementation of this detector are displayed below:

- `ir.UnaryOperation` - IR node that represents any unary operation. It has as attributes two following IR nodes:
  - `_sub_expression: ExpressionAbc` - represents an expression on which operation is executed;

    it is worth to mention that every `ExpressionAbc` object has attribute `types.TypeAbc`, which can identify the type of variable.
  - `_operator: UnaryOpOperator` - represents operation, that executes on expression;

  The algorithm for finding this error is straightforward:

1. Iterate by every `delete` operation, which is represented in the IR tree as `ir.UnaryOperation` node;
2. Check if the deleted type has mapping, which is nested to the struct;

This detector implementation is displayed below:

```python
class StructMappingDeletionDetector(Detector):
    _detections: List[DetectorResult]

    def __init__(self) -> None:
        self._detections = []

    def detect(self) -> List[DetectorResult]:
        return self._detections

    def _check_struct_mapping(
        self,
        t: types.TypeAbc) -> Set[ir.VariableDeclaration]:
        if instance(t, types.Array):
            return self._check_struct_mapping(t.base_type)
        elif isinstance(t, types.Struct):
            ret = set()
            for m in t.ir_node.members:
                if isinstance(m.type, types.Mapping):
                    ret.add(m)
                else:
                    ret.update(self._check_struct_mapping(m.type))
            return ret
        else:
            return set()

    def visit_unary_operation(self, node: ir.UnaryOperation):
        if node.operator != ir.enums.UnaryOpOperator.DELETE:
            return

        t = node.sub_expression.type
        assert t is not None
        members = self._check_struct_mapping(t)
        if len(members) > 0:
             # FINDING
            pass
```

The function `visit_unary_operation` iterates over all `ir.UnaryOperation` nodes and check if the operation is the `delete`. After that, the algorithm verifies whether the deleted type includes a nested mapping within the struct.

2.4.3 'msg-data in Keccak' detector

Randomness is crucial in smart contracts due to security reasons. In contexts such as decentralized finance (DeFi) [29], randomness serves as an essential mechanism for allocating roles, resources, or rewards. Predictability in such cases could lead to vulnerabilities where an attacker could manipulate outcomes to their benefit, potentially stealing assets or corrupting the system. Besides, randomness is used to ensure that outcomes cannot be anticipated beforehand. In various decentralized applications, this unpredictability is essential to maintain trust and integrity within the system.

The Keccak hash function, part of the SHA-3 family, is commonly used in blockchain technologies, including Ethereum, for generating pseudo-random numbers within smart contracts. The following points explain why it is important to pass true random data to the Keccak hash function:

- **Prevent Exploitation**. If the input to the Keccak function is predictable, then the output also becomes predictable. This can lead to exploitation, where an attacker might predict the next state of a contract or the result of a transaction before it occurs.

- **Avoid Manipulation**. Smart contracts often use events or states that involve financial transactions. If the output of the Keccak function (used, for instance, in a lottery smart contract) can be influenced by manipulating the input data, it can result in financial loss or unfair advantages.

The primary purpose of the 'msg-data in Keccak' detector is to identify the use of predictable data, specifically `msg.data`, which represents the data passed with a call to a smart contract. Since the Keccak function is open-source, anyone can use it to predict the hash generated from `msg.data`. As a result, using Keccak in this manner undermines its effectiveness. The simplest example of a code snippet where the detector needs to detect the issue:

```
function badRandomness() pure public returns(bytes32) {
    bytes32 upredictable_data = keccak256(msg.data);
    return upredictable_data;
}
```

In this code block, `msg.data` is used as a single parameter for the Keccak hash function, eliminating any randomness.

2.4.3.1 Design of the detector

Data Dependency Graph (DDG) and Control Flow Graphs (CFG) are integral parts of the implementation of this detector. To address the security vulnerability arising from the predictable usage of the `msg.data` parameter in the Keccak hash algorithm, our approach identifies relevant nodes in the DDG representing:

- **DdgNode**. Data from `msg.data`;
- **DdgNode**. Hash created by the Keccak function.

This identification is done by the helper functions available within the Wake framework.

The algorithm for finding this security issue is straightforward but resource-consuming compared to others.

1. The detector takes previously found **DdgNode**s, that represents `msg.data` and Hash created by the Keccak function. As a result, we have only two nodes, which collect all occurrences of `msg.data` and Hash data from Keccak.
2. The detector finds all paths in DDG from `msg.data` node to the hash data node.
3. The detector checks if at least one path from DDG can be reproduced in any CFG or, in other words, in any function of the smart contract.
4. The detector reports a detailed summary of the found security issue.

Due to the detector's closed-source manner, no code can be disclosed in this thesis. However, the archive with the explained detectors is given to the commission. As a result, the code of this detector can be found there.

2.4.3.2 Detector's problems

During analysis of this detector, it has been found that it cannot identify the problem if `msg.data` variable is passed to any global function, and the output of this function is given to the Keccak hash function. The simplest example of code, where the detector does not identify the issue, can be seen below:

```
function badRandomness() pure public returns(bytes32) {
    bytes32 hash_data = keccak256(abi.encode(msg.data));
    return hash_data;
}
```

Chapter 3

Performance evaluation

This chapter describes the evaluation of existing detectors introduced in the previous chapter. In the beginning, it defines metrics and approaches for its evaluation. After that, the results of the measurement are introduced.

3.1 Evaluation system design

We decided to perform two different experiments to evaluate the detectors' effectiveness. The first experiment will be on the created test suites. The second experiment will be on real smart contracts on the different chains used nowadays. In the evaluation of the effectiveness of the implemented Wake detectors, the following attributes are used:

- **Precision**. The positive predictive value measures the ratio of predicted true positives among all instances classified as positive.

$$\text{Precision} = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalsePositive}} \tag{3.1}$$

- **Recall**. Also known as the true positive rate, it calculates the proportion of correctly predicted True Positives out of all False Negatives.

$$\text{Recall} = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalseNegative}} \tag{3.2}$$

- **F1-Score**. The harmonic mean between precision and recall.

$$\text{F1-Score} = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.3}$$

As the Wake framework static analysis tool has a decent amount of detectors, it is impossible to evaluate them all correctly. It has been decided to evaluate only detectors described in the previous chapter, such as 'Call options not called' and 'Struct mapping deletion' detectors.

3.2 Measurements

3.2.1 First experiment

The first test suite for the detector 'Call options not called' is composed of different variants of calls. It contains:

- new version of 'call' syntax;
- old version of 'call' syntax;
- combination of old and new version of 'call' syntax;
- call using interfaces.

The second test suite is written explicitly for 'Struct mapping deletion', consisting of comprehensive nested structures with mappings.

The third test is defined for testing the performance of the 'msg-data in Keccak' detector. It consists of simple tests and nested tests.

The test shows that implemented detectors succeeded in defining almost every True Positive and every Negative Positive. These results have been achieved with the help of a structural approach and not a heuristic. The used test suite can be found in the archive, which is attached to this thesis. However, it should be mentioned that this detector solves a trivial and straightforward task. More difficult implementations of the detector will be introduced in the next chapter.

Table 3.1 Performance Metrics

Name of the detector	Precision	Recall	F1-Score
Call options not called	100%	100%	100%
Struct mapping deletion	100%	100%	100%
msg-data in Keccak	100%	90%	94,74%

3.2.2 Second experiment

A total of 500,508 contracts, sourced from various blockchain networks such as the Binance Chain and Ethereum Chain, have been selected for the evaluation of specific detectors, including 'Call options not called,' 'Struct mapping deletion,' and 'msg-data in Keccak'. Due to time constraints, it was not feasible to quantify the prevalence of False Negatives within these contracts. As a result, the analysis was restricted to the assessment of True Positives and False Positives.

The results collected from this experiment are remarkable, given that the underlying complexities of the issues these detectors address are not difficult.

Table 3.2 Performance Metrics for the Second experiment

Name of the detector	True Positive	False Positive	Number of findings
Call options not called	100%	0%	86
Struct mapping deletion	100%	0%	8
msg-data in Keccak	100%	0%	193

The simple nature of the problem has led to the creation of detector designs that are straightforward yet highly effective in their performance

These detectors demonstrate remarkable precision, as evidenced by the results, which confirm their capability to accurately identify only those issues that are True Positives.

3.3 Conclusion of existing detectors' analysis

These detectors have been selected as the most representative examples of analyzers implemented using the Wake framework. They distinctly illustrate the unique advantages of detectors developed through a structured approach, specifically their ability to accurately identify security issues without flagging non-existent problems. Developers of Wake have tried to develop tools that facilitate both high-quality and structured code analysis. Tools like Data Dependency and Control Flow Graphs are the most powerful tools in the Wake framework, and they can be used for designing and implementing sophisticated detectors.

Chapter 4

Implementation of the detector

This chapter describes a development process if the detector 'write-after-write' has. Firstly, it introduces the problem this detector intends to solve. Secondly, it proposes the design of this detector. Thirdly, the implementation of the detector using the Wake framework is stated. Lastly, it depicts challenges that have been encountered during the development process.

4.1 Problem 'write-after-write'

The main problem that this detector aims to solve is to find writing to a variable value that will not used after that in the code. A simple example of the problem is presented below:

```
// FINDING
function simpleFunc() public returns (uint){
    uint a;
    a = 42;
    a = 1337;
    return a;
}
```

This code snippet depicts the variable `a` that has been overwritten two times, but only the last value assigned to it has been somehow used. The problem is that the first assigned value to the `a` will not be used. Firstly, it is not suitable for user experience and contracts because almost every operation in the EVM costs gas, and whoever uses this function will need to pay an additional price for the gas. Secondly, it can indicate that a value is not used in the code after being previously assigned, which is an even bigger problem and can create security issues.

However, it should be mentioned which cases are not the issues. When ’write-after-write’ occurs only in specific cases, it is not the problem.

```
// NOT FINDING
function simpleFunc(uint256 b ) public returns (uint){
    uint a;
    a = 42;
    if (b == 100){
        a = 1337;
    }
    return a;
}
```

The main principle for developing this detector is to make it as accurate as possible. We have decided to implement this detector with the following main idea: ”When at least one path to use the assigned value for the variable exists - it is not considered an issue.”

4.2 ’write-after-write’ detector

4.2.1 Design of the detector

The main goal of the detector is to be as precise as possible and, ideally, not to generate False Positives at all. The analysis will primarily be based on nodes and edges of DDG, where the detector examines all writes to the variable and all its usages, called ’reads’ of the variable in another way, in the code. However, defining the order in which writes and reads are executed is impossible without analyzing CFG. Therefore, this tool will also be used.

4.2.1.1 Algorithm to find an issue

Every write operation for every variable will be analyzed. The detector will distinguish between *first write operation* under analysis and a list of *other write operations* on this variable. If the situation where *first write operation* will not be used in any existing path of CFG because of being overwritten by another write operation from *other write operations* list occurs, the detector will highlight that the first write operation is not used and have been overwritten. If put more simply, this algorithm can be described using a modified child game called ’Tag’. In this game, children are divided into two groups. One group is supposed to come to the exit, and the second group tries to catch them. In our case, the member of the first group is a single element from *first write operations*. The second group consists of elements from *other write operations* group, which are also connected with the same variable as the *first write operation*, which is now under analysis. The exit, where element from the first group aims to come, are read operations and end blocks of the CFG. The game field, which, in this case, is all possible paths, is provided by CFG. In other words, we can describe the overview of this algorithm in the following steps:

1. Collect all variables by using a Data Dependency Graph with type `VariableDdgNode` that are not state variables.

2. For variable, find all the edges connected to and out of this variable in the Data Dependency Graph.

3. Find every representation statement in the Control Flow Graph for the Data Dependency Graph edge. (Figure 3.3)

4. Divide these statements by different Control Flow Graphs.

5. Analyze every write operation of this variable, as it is *first write operation*. Collect other writes, previously named as *other write operations*, which connects with *first write operation*. In addition, collect all read operations on the variable and end node of the CFG. This list will present the possible exits. Find out if at least one path by which *first write operation* can come to any elements from the exit list bypassing all *other write operations* exist. If possible, then *first write operation* is safe, and no finding is reported. If no path exists, the 'write-after-write' for a particular analyzed *first write operation* is reported. Algorithm goes to step **2**.

The algorithm finishes at the moment when all non-state[1] variables of the smart contract have been analyzed. The simplified scheme of the algorithm is introduced in Figure 3.4.

4.2.2 Challenges during implementation

A few challenges have appeared during implementation. Part of these challenges were due to Solidity features, and another part was due to the insufficiently deep implementation of built-in tools in the Wake framework.

4.2.2.1 Function arguments

Detector has to to process variables that specially are function arguments, as one write operation already has been performed on them. For example, if the detector analyzes these variables in the same way as common variables, there will be no finding in the code below:

```
function simpleFunc(uint a) public returns (uint){
    a = 42;
    return a;
}
```

[1]state variable - a variable whose value is permanently stored in contract storage, for example, attribute of smart contract

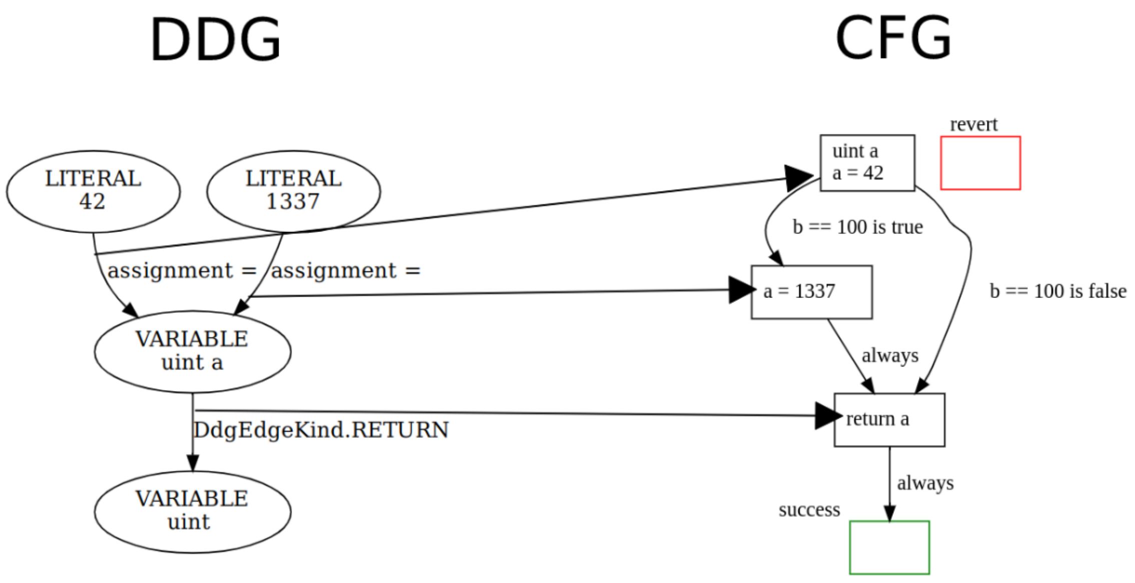


Figure 4.1 Extracting information from CFG using DDG

However, if it is a function argument, it already has passed value due to a function call. Thus, we need to add one write operation to the start block of the function CFG. In this case, an additional write operation will always be the first in the CFG of the called function.

4.2.2.2 Function arguments and modifiers

There is a possibility that passed function arguments will be used by the modifier of this function, as in the example below:

```
function simpleFunc(uint a) bar(a) public returns (uint){
    a = 42;
    return a;
}
```

If this is the case, we need to add one additional read operation to every variable from function arguments used by modifiers.

4.2.2.3 Representation of shortcut for assignments in DDG

During process development, it has been found that the Data Dependency Graph created by the Wake framework does not correctly represent assignment shortcuts. The operation `i += 1` only creates one write operation to the variable `i` in DDG. However, a hidden read operation of `i` variable exists

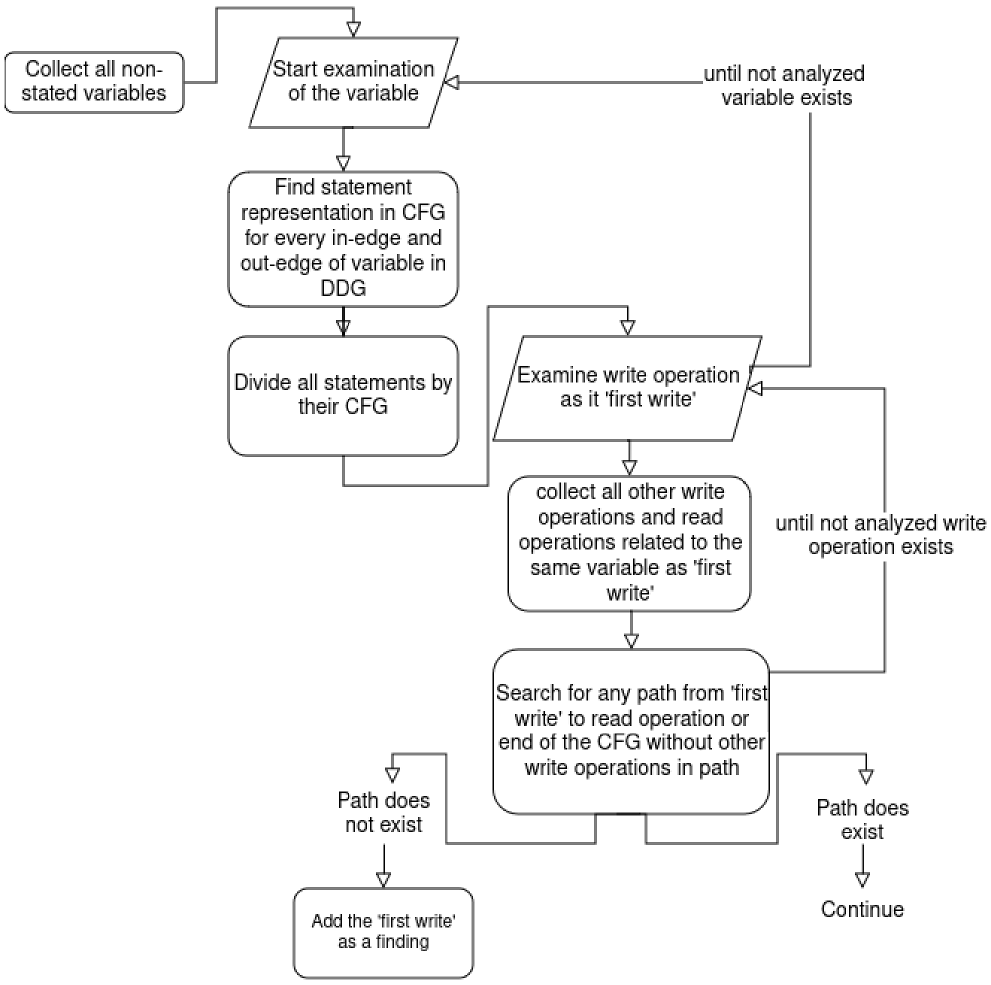

Collect all non-stated variables
Start examination of the variable
until not analyzed variable exists
Find statement representation in CFG for every in-edge and out-edge of variable in DDG
Divide all statements by their CFG
Examine write operation as it 'first write'
collect all other write operations and read operations related to the same variable as 'first write'
until not analyzed write operation exists
Search for any path from 'first write' to read operation or end of the CFG without other write operations in path
Path does not exist
Path does exist
Add the 'first write' as a finding
Continue

Figure 4.2 Algorithm to find write-after-write

before the write operation because shortcut `i += 1` stands for `i = i + 1`. For instance, the described problem can be seen in the code below and in Figure 3.5, which represents the DDG graph of the variable `i`:

```
function simpleFunc() public {
    uint i;
    i = 0;
    i += 1;
}
```

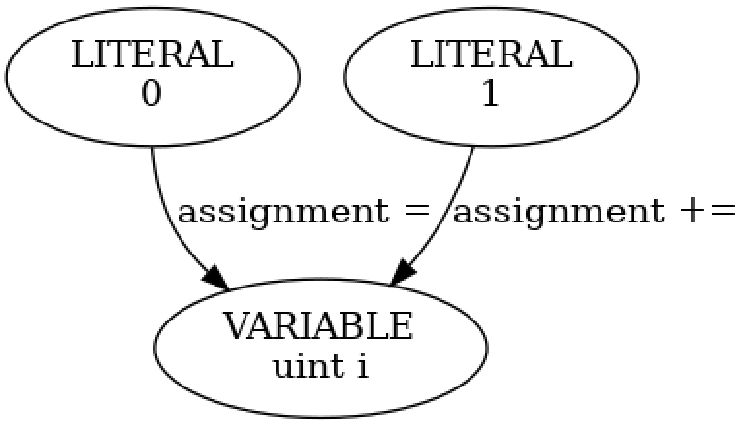


Figure 4.3 The DDG graph of the i variable

As can be noticed, there is no edge from the `i` variable. In this case, an additional edge from the `i` variable to itself has to be added to the implementation code.

4.2.2.4 Representation of loops in CFG

In the Wake framework, it was discovered that the control statement for loop iteration is stored not within a Control Flow Graph (CFG) node but instead on an edge. This positioning means the control statement doesn’t appear in the list of a block’s statements. Consequently, any analysis based on the list of statements within CFG nodes may overlook the loop control mechanisms, potentially impacting the accuracy of the analysis. An example of the function that replicates this behavior and its CFG graph is introduced below:

```
function simpleFunc() pure public returns (uint){
    uint b;
    b = 10;
    for (int a = 0; a < 10; a++){
        b = b + 1;
    }
}
```

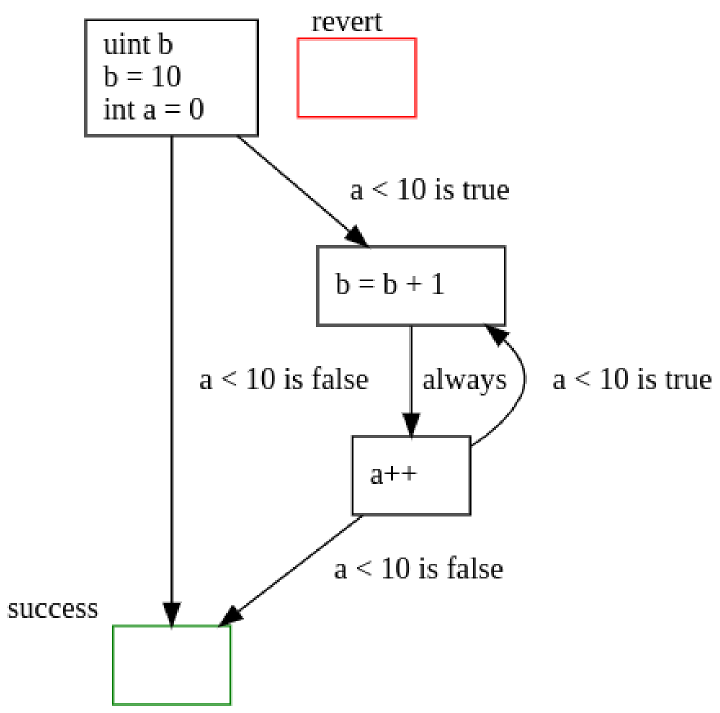


Figure 4.4 The CFG graph of the “simpleFunc” function

Wake framework’s The Control Flow Graph (work uniquely stores control statements. These statements are positioned on the edges rather than within the nodes of the CFG, which differs from typical CFG implementations. This distinctive architecture requires an additional step in the analysis process. For accurate analysis, it is essential to integrate the control statements from the edges into the corresponding CFG node blocks. By manually appending these control statements to the end of node blocks from which the edges originate, we can remedy the gap and maintain the integrity of the analysis.

4.2.3 Unsolved challenges

During the implementation process, we found problems that could not be solved by simply modifying the algorithm logic. A new approach to solving them must be chosen.

4.2.3.1 State variable problem

In smart contracts, there is no one main execution flow. A smart contract is a set of functions that can be called and used. A state variable is a variable that is saved in a smart contract as its attribute, so it can be changed not only by the called function but also by another function of the same smart contract that the called function uses. Coming out of all of the above, for analyzing a state variable, we have to analyze all possible sequences of smart contract functions that can somehow influence a state variable. Thus, the introduced algorithm for finding ’write-after-write’ issues cannot be used for detecting the same issue for the state variables. The code below demonstrates a simple example of why it is not possible to analyze state variables in the same way:

```
contract BContract{
    unit a;
    bool notEntered;
    function foo(bool notEntered) public {
        if (notEntered == false){
            a = 42;
        }
    }
    modifier bar(uint b){
        notEntered = false;
        _;
        notEntered = true;
    }
    function simpleFunc(uint b) bar(b) public {
        foo(notEntered);
    }
}
```

The modifier function `bar()` performs two write operations on the variable `notEntered`. The algorithm previously described might flag this as an issue. However, such a flag would be incorrect because a function that utilizes the `notEntered` state variable is called between these write operations. The Control Flow Graph (CFG) of the modifier function does not detect the read operation inside the other function. This oversight occurs because the state variable `notEntered` exists within the scope of the smart contract, not just within a single function.

4.2.3.2 Assignments in the loop

The detector has to identify all unnecessary or forgotten write operations. However, there is an example where the write operation wily be unnecessary only in the specific path. A simple example of it is introduced in the code below:

```
function simpleFunc() pure public {
    uint a;
    for (int i = 0; i < 10; i++){
        a = 10;
    }
}
```

While looking into the code, it can be seen that the assignment operation for `a` variable can be executed after the loop, as the operation assigns the same value every iteration. Nevertheless, there is only one possible path in the CFG where the flow of the program can go. Due to the compilation phase, there is no possibility of finding out that the `for` loop will be executed exactly ten times. As a result, the designed algorithm has it is not identified as an issue.

4.2.4 Future work

Several improvements could be made to the detector to extend its capability to identify a broader range of issues. These improvements would involve refactoring the underlying algorithm and expanding the scope of conditions that the detector monitors.

4.2.4.1 Structure compatibility

Detector can be improved by taking into account structure objects. For instance, it can analyze simple attributes of the structure objects and find issues connected with objects. As a result, the detector can analyze more complex data structures. The code that can create an issue is located below.

```
contract BContract {
    struct S {
        uint x;
    }

    function foo(S memory s) pure public {
        s.x = 5;
        s.x = 6;
    }
}
```

Here, the issue occurred due to overwriting the value of the structure object’s attributes `s.x` without using the first assigned value.

4.2.4.2 Detect assigning of the default value

Developers may occasionally assign a default value to a variable despite its unnecessary nature, as variables declared in the Ethereum Virtual Machine (EVM) inherently have a zero value. Technically, it is not a security issue but a code style. However, the designed algorithm doesn’t consider it for now, and assigning the default value is an issue. The code snippet below introduces the typical example of the described problem:

```
function simpleFunc(uint b) pure public {
    uint a = 0;
    if (b == 10){
        a = 42;
    }else {
        a = 43;
    }
}
```

The detector can be modified to ignore the default value assigned during variable declaration, treating it as a non-write operation. This prevents any issue arising from default value assignment from being detected.

Chapter 5

Testing & Results

This chapter describes a performance evaluation of the implemented 'write-after-write' detector and compares it with the Slither 'write-after-write' detector. The system to evaluate the detector's performance is defined in the first chapter. The second section introduces the results of the experiments, which compare the results of the two implementations on the prepared set of contracts. After that, the second experiment is shown, where both detectors have been tested on an extensive set of vulnerable smart contracts.

5.1 Testing methodology

Evaluating a detector's performance using static code analysis to find security issues is crucial to ensuring the accurate 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.

5.1.1 Evaluating the result from the first experiment

As the limited number of prepared smart contracts need to be audited, they can be analyzed manually. Therefore, we can assess the evaluator using them more precisely. The following metrics are used:

- **Precision**
- **Recall**
- **F1-Score**

Given that the above-mentioned metrics were already described in Chapter 3, reiterating them is unnecessary.

5.1.2 Evaluating the result from the second experiment

The already analyzed dataset by Slither detector is taken for analyzing. [30] Due to the analysis of approximately 10,000 smart contracts in the second experiment, manual inspection of each one is impractical. Therefore, to manage time effectively, 50 flagged smart contracts from each detector were randomly selected for evaluation. Besides, a set of contracts in which Slither has detected the issue and Wake has not found, were analyzed. The following metrics were chosen for assessment:

- **False Positive amount**;
- **True Positive amount**;
- **Precision**;
- **Number of found issues**

These metrics are well-suited for conducting comprehensive analyses while maintaining quality standards. By using these measures, we can ensure robust evaluation outcomes that accurately reflect the performance and effectiveness of the detectors.

5.2 Performance evaluation

5.2.1 First experiment

Exactly 34 prepared smart contracts have been 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.

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:

Table 5.1 Performance Evaluation

	Slither	Wake
Precision	$\frac{7}{7+0} = 100\%$	$\frac{16}{16+0} = 100\%$
Recall	$\frac{7}{7+14} = 33.33\%$	$\frac{16}{16+5} = 76.19\%$
F1-Score	$\frac{2*100\%*33.33\%}{100\%+33.33\%} = 50\%$	$\frac{2*100\%*76.19\%}{100\%+76.19\%} = 86,49\%$

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

5.2.2 Second experiment

This investigation selected a subset of vulnerable smart contracts from the dataset by Rossini (2022) [31]. 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. 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;
- Sent to Etherscan samreen2021smartscan smart contract uses absolute paths.

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:

Table 5.2 Performance Metrics for Slither's additional findings

Metrics	True Positive	False Positive	Precision
Slither	10	53	15.9 %

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 amount found by the realization of the detector using the Wake framework is bigger. As seen by previous and further statistics, the Wake detector has better precision, which is close to 100 percent.

Table 5.3 Number of found issues by Wake and Slither

Metrics	Number of findings
Slither	199
Wake	**672**

The following table provides an overview of the analysis, for which 50 smart contracts have been randomly chosen and manually analyzed. Thus, we did not identify any 'write-after-write' issue in 12 contracts, and 38 contracts have been flagged as having security issues.

Table 5.4 Performance Evaluation between Slither and Wake on example 'write-after-write' detector

	Slither	Wake
Precision	$\frac{19}{19+3} = 86.36\%$	$\frac{36}{36+0} = 100\%$
Recall	$\frac{19}{19+19} = 50\%$	$\frac{36}{36+2} = 94.74\%$
F1-Score	$\frac{2*86.36\%*50\%}{50\%+86.36\%} = 63.33\%$	$\frac{2*100\%*94.74\%}{100\%+94.74\%} = 97.29\%$

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.

5.3 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 5.1, Table 5.2, and Table 5.4, the Wake detector demonstrates greater precision than the Slither detector.
- Based on results from Table 5.3, the Wake detector identifies more security issues than the Slither detector.

Conclusion

The first goal of the thesis was to analyze the design of already implemented detectors – the second chapter presented the design and implementation of the existing detectors: 'Call options not called,' 'Struct mapping deletion', and 'msg-data in Keccak'. Besides, the Intermediate Representation (IR) model of the Wake framework used in these detectors has been described in the design discussion.

The second goal was to evaluate existing detectors on a provided set of contracts. In the third chapter, the evaluation system design was defined, and measurements were done on a prepared set of smart contracts hosted on real production chains, such as the Ethereum mainnet.

The third goal was to implement a vulnerability detector using the Wake framework. The fourth chapter describes the process of developing the new 'write-after-write' detector using the Wake framework. Besides, it introduces the challenges met in the development process and how they have been solved. In addition, ideas on how the detector can be improved are introduced.

The fourth goal was to evaluate the performance of the implemented detector compared to the same detector of a state-of-the-art analysis tool. In chapter five, an evaluation of the implemented 'write-after-write' detector using the Wake framework and the existing 'write-after-write' implemented by Slither was done. Therefore, it has been proved statistically that the implemented detector outperforms the existing Slither detector, which addresses the same security issue.

This thesis provides enough information about the static analysis vulnerability detector's implementation processes; hence, every reader interested in designing their own detector could implement a simple one and improve the overall security of the blockchain environment.

Bibliography

1. LLC, Google. *How the word 'cryptocurrency' is popular* [online]. 2024. [visited on 2024-04-24]. Available from: `https://books.google.com/ngrams/graph?content=cryptocurrency&year_start=1800&year_end=2019&corpus=en-2019&smoothing=3`.
2. ZHENG, Zibin; XIE, Shaoan; DAI, Hong-Ning; CHEN, Xiangping; WANG, Huaimin. Blockchain challenges and opportunities: A survey. *International journal of web and grid services*. 2018, vol. 14, no. 4, pp. 352–375.
3. CHAINALYSIS. *Stolen Crypto Falls in 2023, but Hacking Remains a Threat* [online]. 2024. [visited on 2024-04-24]. Available from: `https://www.chainalysis.com/blog/crypto-hacking-stolen-funds-2024/`.
4. DANNEN, Chris; DANNEN, Chris. Solidity programming. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. 2017, pp. 69–88.
5. NAKAMOTO, Satoshi. *Bitcoin: A Peer-to-Peer Electronic Cash System* [White paper]. 2008. Available also from: `https://bitcoin.org/bitcoin.pdf`.
6. LASHKARI, Bahareh; MUSILEK, Petr. A comprehensive review of blockchain consensus mechanisms. *IEEE access*. 2021, vol. 9, pp. 43620–43652.
7. BÖHME, Rainer; CHRISTIN, Nicolas; EDELMAN, Benjamin; MOORE, Tyler. Bitcoin: Economics, technology, and governance. *Journal of economic Perspectives*. 2015, vol. 29, no. 2, pp. 213–238.
8. LIU, Debin; CAMP, L Jean. Proof of Work can Work. In: *WEIS*. Citeseer, 2006.
9. VUJIČIĆ, Dejan; JAGODIĆ, Dijana; RANĐIĆ, Siniša. Blockchain technology, bitcoin, and Ethereum: A brief overview. In: *2018 17th international symposium infoteh-jahorina (infoteh)*. IEEE, 2018, pp. 1–6.

10. WOOD, Gavin et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*. 2014, vol. 151, no. 2014, pp. 1–32.

11. OMAR, Ilhaam A; JAYARAMAN, Raja; DEBE, Mazin S; HASAN, Haya R; SALAH, Khaled; OMAR, Mohammed. Supply chain inventory sharing using ethereum blockchain and smart contracts. *IEEE access*. 2021, vol. 10, pp. 2345–2356.

12. SRIMAN, B; KUMAR, S Ganesh. Decentralized finance (defi): the future of finance and defi application for Ethereum blockchain based finance market. In: *2022 International Conference on Advances in Computing, Communication and Applied Informatics (ACCAI)*. IEEE, 2022, pp. 1–9.

13. ETHEREUM DEVELOPERS. *Intro to Ethereum*. 2024. Available also from: `https://ethereum.org/en/developers/docs/intro-to-ethereum/`. [Online; accessed 2024-04-02].

14. WOHRER, Maximilian; ZDUN, Uwe. Smart contracts: security patterns in the ethereum ecosystem and solidity. In: *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2018, pp. 2–8.

15. ETHEREUM DEVELOPERS. *The "Yellow Paper": Ethereum's Formal Specification*. 2024. Available also from: `https://ethereum.github.io/yellowpaper/paper.pdf`. [Online; accessed 2024-04-05].

16. BUTERIN, Vitalik et al. A next-generation smart contract and decentralized application platform. *white paper*. 2014, vol. 3, no. 37, pp. 2–1.

17. ETHEREUM DEVELOPERS. *Gas and Fees*. 2024. Available also from: `https://ethereum.org/en/developers/docs/gas/`. [Online; accessed 2024-04-05].

18. DEVELOPERS, Solidity. *Introduction to Smart Contracts* [`https://docs.soliditylang.org/en/v0.8.25/introduction-to-smart-contracts.html#storage-memory-and-the-stack`]. 2022. Accessed: [2024-04-05].

19. ETHEREUM FOUNDATION. *EVM architecture* [Online]. 2024. [visited on 2024-04-12]. Available from: `https://ethereum.org/content/developers/docs/gas/gas.png`. Available from: `https://ethereum.org/content/developers/docs/gas/gas.png`.

20. GAITAN, Vasile Gheorghita; GAITAN, Nicoleta Cristina; UNGUREAN, Ioan. CPU architecture based on a hardware scheduler and independent pipeline registers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. 2014, vol. 23, no. 9, pp. 1661–1674.

21. ZOU, Weiqin; LO, David; KOCHHAR, Pavneet Singh; LE, Xuan-Bach Dinh; XIA, Xin; FENG, Yang; CHEN, Zhenyu; XU, Baowen. Smart contract development: Challenges and opportunities. *IEEE transactions on software engineering*. 2019, vol. 47, no. 10, pp. 2084–2106.
22. ETHEREUM DEVELOPERS. *Solidity*. 2024. Available also from: `https://docs.soliditylang.org/en/v0.8.25/`. [Online; accessed 2024-05-14].
23. GRAMLICH, Benjamin. Smart contract languages: A thorough comparison. *ResearchGate Preprint*. 2020, pp. 1–6.
24. SOLIDITY TEAM. *Contracts* [`https://docs.soliditylang.org/en/v0.8.24/contracts.html`]. 2024. Accessed on 2024-04-06.
25. BISTARELLI, Stefano; MAZZANTE, Gianmarco; MICHELETTI, Matteo; MOSTARDA, Leonardo; TIEZZI, Francesco. Analysis of ethereum smart contracts and opcodes. In: *Advanced Information Networking and Applications: Proceedings of the 33rd International Conference on Advanced Information Networking and Applications (AINA-2019) 33*. Springer, 2020, pp. 546–558.
26. ALCHEMY. *Solidity Compiler Overview* [`https://www.alchemy.com/overviews/solidity-compiler`]. 2024. Accessed: [Insert Access Date Here].
27. ZHU, Nicole. *Ethernaut Lvl 19 MagicNumber Walkthrough: How to Deploy Contracts Using Raw Assembly Opcodes* [`https://medium.com/coinmonks/ethernaut-lvl-19-magicnumber-walkthrough-how-to-deploy-contracts-using-raw-assembly-opcodes-c50edb0f71a2`]. 2024. Accessed: [Insert Access Date Here].
28. MØLLER, Anders; SCHWARTZBACH, Michael I. Static program analysis. *Notes. Feb*. 2012.
29. ZETZSCHE, Dirk A; ARNER, Douglas W; BUCKLEY, Ross P. Decentralized finance (defi). *Journal of Financial Regulation*. 2020, vol. 6, pp. 172–203.
30. MWRITESCODE. *Slither Audited Smart Contracts* [`https://huggingface.co/datasets/mwritescode/slither-audited-smart-contracts`]. 2024. Accessed on [Insert Access Date Here].
31. ROSSINI, Martina. *Slither Audited Smart Contracts Dataset*. 2022.

Content of the attachment

```
khimcdmy-thesis.pdf ..................... Khimchenko Bachelor Thesis
├── addresses_analyzed .................. Used Addresses During Analysis
│   ├── 50_addresses_manually_analyzed.txt
│   ├── addresses_detected_by_slither_but_not_wake.txt
│   └── all_addresses_analyzed.txt
├── contracts ........................... Test-Suite Contracts
│   ├── test_suite_call_options_not_called.sol
│   ├── test_suite_mapping_into_struct_deletion.sol
│   ├── test_suite_msg_data.sol
│   └── test_suite_write_after_write.sol
├── detectors.zip ....................... Close-Sourced Detectors
│   ├── write_after_write.py
│   ├── msg_data.py
│   └── utils
├── thesis
└── README.md ........................... Description of Directory
```