FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Financial impact of Ethereum vulnerability detectors
Student:	Bc. Andrey Bortnikov
Supervisor:	Ing. Josef Gattermayer, Ph.D.
Study program:	Informatics
Branch / specialization:	Managerial Informatics
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2024/2025

Instructions

In DeFi and other areas where the Ethereum blockchain is used to store value, 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. The cost of attacks prevented by detectors can be easily calculated.

Instructions:
- Analyze existing Wake detectors, including the IR data model used in the detectors.
- Define a testing set of mainnet contracts holding token values, calculate the total value.
- Evaluate the performance of the existing detectors on a set of contracts, evaluate the cost of possible exploits prevented by the detectors.
- Design and implement new bug and vulnerability detectors in consultation with the supervisor.
- Test the implemented detectors on the set of contracts, evaluate the cost of possible exploits prevented by the implemented detectors.

Electronically approved by Ing. David Buchtela, Ph.D. on 25 January 2024 in Prague.

Master's thesis

FINANCIAL IMPACT OF ETHEREUM VULNERABILITY DETECTORS

Bc. Andrey Borntikov

Faculty of Information Technology
Software Engineering Department
Supervisor: Ing. Josef Gattermayer, Ph.D.
May 8, 2025

Contents

List of Figures

List of Tables

List of code listings

I thank my family and friends for their help and support throughout my studies. I also thank Ing. Josef Gattermayer, Ph.D., for being my supervisor and CTU for the kind acceptance and high-quality teaching.

Declaration

Abstract

This Master's thesis investigates the financial implications of vulnerability detectors within Ethereum, a prominent blockchain platform. A quantitative approach is utilized to analyze a comprehensive set of Ethereum smart contracts, particularly emphasizing those that are recent, have substantial liquidity, and have notable popularity. The findings reveal a significant correlation between the detection of vulnerabilities and financial losses, with undetected vulnerabilities accounting for a substantial proportion of these losses. The study concludes that enhancements in the accuracy and efficiency of Ethereum vulnerability detectors could lead to significant annual financial savings. This research highlights the critical role of vulnerability detectors in protecting digital assets and maintaining financial stability within blockchain ecosystems.

Keywords Ethereum, Vulnerability Detectors, Smart Contracts, Financial Implications, Blockchain Security, Digital Assets, Financial Stability, Artificial Intelligence

Abstrakt

Tato diplomová práce zkoumá finanční důsledky detektorů zranitelností v rámci Ethereum, významné blockchainové platformy. K analýze uceleného souboru smart kontraktů Ethereum je využit kvantitativní přístup, přičemž je kladen důraz zejména na ty, které vznikly nedávno, mají značnou likviditu a vyznačují se pozoruhodnou popularitou. Zjištění odhalují významnou korelaci mezi detekcí zranitelností a finančními ztrátami, přičemž neodhalené zranitelnosti tvoří podstatnou část těchto ztrát. Studie dochází k závěru, že zvýšení přesnosti a účinnosti detektorů zranitelností Ethereum by mohlo vést k významným ročním finančním úsporám. Tento výzkum zdůrazňuje zásadní roli detektorů zranitelností při ochraně digitálních aktiv a udržování finanční stability v blockchainových ekosystémech.

Klíčová slova Ethereum, detektory zranitelnosti, smart kontrakty, finanční dopady, bezpečnost blockchainu, digitální aktiva, finanční stabilita, umělá inteligence

List of abbreviations

API	Application Programming Interface
ETH	Ethereum
EVM	Ethereum Virtual Machine
DeFi	Decentralized Finance
CI/CD	Continuous Integration/Continuous Deployment
IDE	Integrated Development Environment
AI	Artificial Intelligence
LLM	Large Language Model
dApp	Decentralized application
AST	Abcstract Syntax Tree
IR	Intermediate Representation
CFG	Control Flow Graph
DDG	Data Dependency Graph
TVL	Total Value Locked
PoW	Proof of Work
PoS	Proof of Stake
NatSpec	Ethereum Natural Specification

Introduction

The security of smart contracts is essential in decentralized finance and other applications utilizing the Ethereum blockchain, where safe value storage is critical. Security vulnerabilities in these contracts can lead to significant financial losses. A practical approach to mitigating these vulnerabilities is to employ static code analyzers during the development lifecycle.

This thesis examines the financial impact of Ethereum vulnerability detectors, particularly their effectiveness in preventing attacks on decentralized applications. We investigate the utility of tools like Wake [1], an open-source platform designed to streamline the development and evaluation of Solidity smart contracts. A key feature of Wake is its module dedicated to detecting vulnerabilities and bugs through static code analysis, which offers opportunities for expanding the tool's capabilities with custom detectors.

In addition to analyzing existing detectors within the Wake framework, a new detector will be implemented and tested. This addition aims to enhance the security measures available for smart contract developers, ensuring that potential exploits are effectively identified and mitigated.

This thesis seeks to provide insights into the effectiveness of vulnerability detectors by analyzing and testing existing and a newly developed detector. Ultimately, the goal is to contribute to a more secure decentralized ecosystem, reducing the likelihood of financial losses due to smart contract vulnerabilities and fostering trust in blockchain technology.

Objectives

The primary objective of this thesis is to analyze and implement solutions that prevent breaches and hacks on the blockchain using static code analysis provided by the Wake framework. The Wake framework, an open-source tool, facilitates the development and auditing of Solidity applications [2]. The specific goals of this thesis are:

- **Analyze Existing Wake Detectors:** Evaluate the detectors, including the Intermediate Representation (IR) data model used for vulnerability detection.
- **Define and Analyze a Testing Set:** Compile a dataset of mainnet smart contracts holding token values and calculate their total value.
- **Evaluate Performance of Existing Detectors:** Assess the effectiveness of existing detectors on the compiled set of smart contracts and evaluate the potential cost of exploits they could prevent.
- **Design and Implement New Detector:** Develop and integrate a new vulnerability detector.
- **Test New Detector:** Test the new detector on the dataset and evaluate the cost of potential exploits.

0.1 Blockchain Introduction

This chapter provides an overview of blockchain technology, focusing on the core principles of distributed ledger technology, including consensus mechanisms, cryptographic foundations, and the nature of blockchain networks. The discussion emphasizes how decentralization ensures security and transparency without centralized oversight.

0.2 Ethereum

Building on the fundamentals of blockchain, this chapter explores Ethereum's programmable architecture. This includes Ethereum's unique features, such as the Ethereum

Virtual Machine (EVM), decentralized applications (dApps), and its native cryptocurrency, Ether. Ethereum allows for programmable functionalities, transforming blockchain technology into a platform suitable for various applications.

Smart contracts are self-executing programs on the Ethereum blockchain that automate and enforce digital agreements. We examine their role and essential security considerations.

Solidity is the primary language used to develop Ethereum smart contracts. We cover its features, syntax, emphasizing crucial aspects such as gas optimization, memory management, and security.

0.3 Wake Analysis

The Wake framework offers advanced tools for Ethereum smart contract development and analysis. This chapter details its static analysis capabilities, particularly the static code analysis, which identifies vulnerabilities and code quality issues.

We examine Wake's current detectors and the Intermediate Representation (IR) data model, focusing on how the IR represents smart contract elements, the types of vulnerabilities it captures, and the effectiveness of current detectors in identifying these vulnerabilities.

0.4 Testing Dataset Compilation

To assess the effectiveness of Ethereum vulnerability detectors within the Wake platform, a toolkit was developed to compile a relevant set of smart contracts, each associated with specific token values. This toolkit helps to automate the collection and enrichment of smart contract data from the Ethereum mainnet for a targeted financial risk analysis.

0.5 Evaluation

This chapter examines the potential cost of exploits based on the value of assets held by the affected contracts. The evaluation focuses on assessing the effectiveness of existing detectors in preventing exploits and quantifying the economic impact of their preventive measures. This estimation considers the value of assets held by the affected contracts.

0.6 New Detector Implementation

This chapter details the development of a new detector for the Wake framework, designed to identify inconsistencies between smart contract documentation and implementation. The detector utilizes artificial intelligence to evaluate code quality and mitigate potential mismatches. Results from testing the detector against real-world contracts are presented.

Assessing the effectiveness of the developed detector on the designated contract portfolio involves conducting an economic evaluation to determine the financial value protected by vulnerability detection.

0.7 Summary

This thesis aims to enhance the security of Ethereum smart contracts through an analysis, development, and evaluation of vulnerability detectors. It improves security measures by implementing and testing a new detector within the Wake framework, reducing financial losses and fostering trust in blockchain technology.

Chapter 1

Blockchain

Blockchain technology represents one of the most significant innovations in distributed computing systems since the advent of the Internet. Before delving into its specific components and mechanisms, it is crucial to understand its fundamental principles and the problems it aims to solve.

1.1 Theoretical Foundations of Distributed Ledger Technology

The concept of distributed ledger technology emerged as a solution to the ongoing challenge of achieving consensus in distributed systems without relying on central authorities for trust. This section explores the theoretical foundations that enable such systems and their practical applications in blockchain technology.

Blockchain technology represents a significant shift in distributed computing systems, changing the architecture of digital transaction systems and data management. This distributed ledger technology utilizes a unique approach to data storage and verification, where information is maintained across a network of participating nodes instead of a centralized repository. The importance of this architecture lies in its ability to uphold data integrity and validate transactions without depending on a central authority.

By examining the historical context and architectural principles that govern blockchain technology, we can gain a clearer understanding of its elements. These insights are essential for grasping the technology's current and future implementation potential. [3]

1.1.1 Historical Context and Evolution

The development of blockchain technology is the result of the convergence of multiple disciplines and decades of research. Understanding its historical progression helps clarify the solutions it offers to fundamental challenges in distributed computing.

The conceptual foundations of blockchain come from years of research in distributed systems, cryptography, and consensus mechanisms. Although the technology gained recognition mainly through its use in cryptocurrencies, its core principles tackle essential issues in distributed computing, such as the Byzantine Generals Problem and the Double-Spending Problem in digital transactions. [4]

1.1.2 Architectural Principles

Having established the historical context, we can now examine the core architectural principles that define blockchain systems. These principles work in concert to create a robust and reliable distributed system.

The blockchain architecture adheres to several fundamental principles:

- **Decentralization**: Distribution of control and validation across network participants.
- **Immutability**: Once recorded, data cannot be altered without network consensus.
- **Transparency**: All transactions are visible to network participants.
- **Cryptographic Security**: Implementation of advanced cryptographic protocols.
- **Consensus-driven**: Network agreement on the state of the ledger.

The implementation of these principles relies on cryptographic foundations, which we will examine in detail in the next section.

1.2 Cryptographic Foundations

The security and reliability of blockchain systems depend on cryptographic principles and their implementation. This section explores the cryptographic components that enable blockchain functionality, beginning with the public key infrastructure that forms the backbone of blockchain security.

1.2.1 Public Key Infrastructure

Public Key Infrastructure in blockchain systems demonstrates a sophisticated implementation of asymmetric cryptographic principles, balancing security with accessibility. The public key components serve three critical functions in the blockchain ecosystem. First, they enable address generation and transaction validation, forming the foundation of secure transfers. Second, they facilitate digital signature verification, allowing network participants to confirm the authenticity of transactions. Third, they establish public identities on the network, creating a transparent yet pseudonymous interaction system.

Private key functionality is the secure counterpart to public key operations. Private keys enable transaction signing, providing cryptographic proof of the owner's intent to transfer assets. They also serve as the definitive proof of ownership for blockchain assets, functioning as a digital form of possession. Additionally, private keys manage access control, determining who can execute specific operations within the blockchain environment.

This dual-key architecture creates a security framework that underpins the entire blockchain ecosystem. It enables trustless transactions and decentralized operations while maintaining high levels of security and verifiability.

1.2.2 Hash Functions in Blockchain

Cryptographic hash functions serve multiple critical purposes in blockchain systems. Their implementation can be represented as:

$$block\_hash = H(previous\_hash \parallel timestamp \parallel nonce \parallel transaction\_data) \tag{1.1}$$

where H represents the hash function, and $\parallel$ denotes concatenation.

The key applications of hash functions in blockchain systems cover several crucial aspects. Data integrity stands as a fundamental application, ensuring that both transaction and block data remain unaltered throughout their lifecycle in the blockchain.

Block linking represents another essential function, creating cryptographic connections between sequential blocks and maintaining the chain's continuity. In systems utilizing Proof of Work, hash functions play a vital role in facilitating mining processes within these consensus mechanisms, providing the computational challenge necessary for secure block validation and chain progression.

1.3 Block Structure and Chain Formation

1.3.1 Block Anatomy

Each block in the blockchain contains several components that can be formally defined as:

$$\text{Block} = \{\text{Header}, \text{Body}\} \tag{1.2}$$

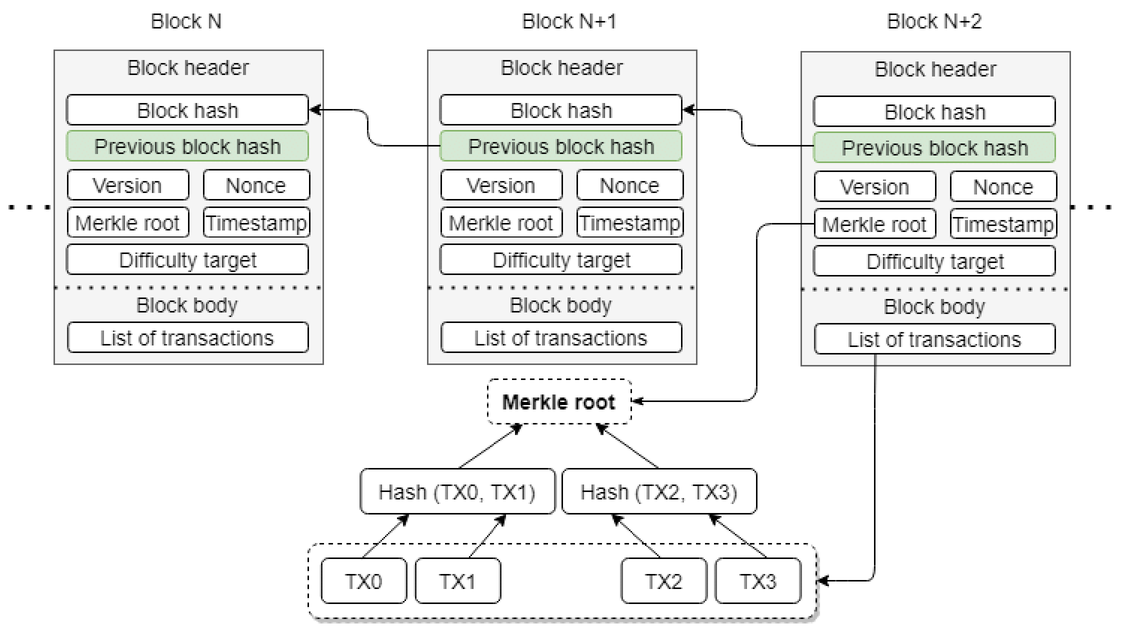
where the **Header** consists of the following fields:

$$\textbf{Header} = \left\{ \begin{array}{ll} \texttt{PrevHash}, & \text{(Hash of previous block)} \\ \texttt{Timestamp}, & \text{(Time block was created)} \\ \texttt{Nonce}, & \text{(Proof of work nonce)} \\ \texttt{MerkleRoot}, & \text{(Merkle root of transactions)} \\ \texttt{Difficulty}, & \text{(Mining difficulty target)} \\ \texttt{Version} & \text{(Block version)} \end{array} \right\} \tag{1.3}$$

The **Body** includes all the individual transactions in the block.

This structure ensures that each block is linked to the previous one, creating a chain of blocks (or blockchain), as shown in Figure 1.1.

Figure 1.1 Blockchain and Block Structure [5]



The blockchain's linear structure is maintained through sequential linking and state transitions. Each block references the previous block's hash, providing a continuous chain of records. State transitions validate state changes, maintain the global state, and verify transaction execution. This mechanism is fundamental in ensuring the integrity and immutability of the blockchain.

1.4 Consensus Mechanisms and Network Security

1.4.1 Proof of Work

Proof of Work (PoW) is one of the earliest and most commonly used consensus algorithms in blockchain technology. It was popularized by Bitcoin and is characterized by its reliance on computational power to validate and secure transactions.

The Proof of Work consensus mechanism implements a computational challenge-response protocol:

```
while True:
    nonce = generate_random_nonce()
    block_hash = hash(block_data + nonce)
    if block_hash < difficulty_target:
        return nonce
```

In PoW, miners (nodes that validate transactions) compete to solve cryptographic puzzles. The puzzle involves finding a nonce such that the resulting hash of the block data, when combined with the nonce, produces a hash value that is below a certain threshold (difficulty target). This process is known as mining.

Key properties and advantages of PoW:

- **Security**: The computational difficulty of mining serves as a barrier against attacks. An attacker would need to control more than 50% of the network's total computational power to successfully execute a double-spending attack.
- **Decentralization**: PoW promotes decentralization by allowing any node with sufficient computational resources to participate in the network.
- **Incentives**: Miners are incentivized through block rewards and transaction fees, ensuring continued participation and security of the network.

Challenges and drawbacks of PoW:

- **Energy Consumption**: PoW requires significant computational resources and energy, leading to high operational costs and environmental concerns.
- **Centralization Risk**: The high cost of mining equipment can lead to centralization, where a few large mining pools control a majority of the network's hash rate.
- **Scalability**: The computational complexity increases over time, making it challenging to scale the network efficiently.

1.4.2 Proof of Stake

Proof of Stake (PoS) is an alternative consensus mechanism designed to address the inefficiencies and environmental impact of PoW. PoS was first proposed in 2011 and has since been implemented in various blockchain platforms, including Ethereum 2.0.

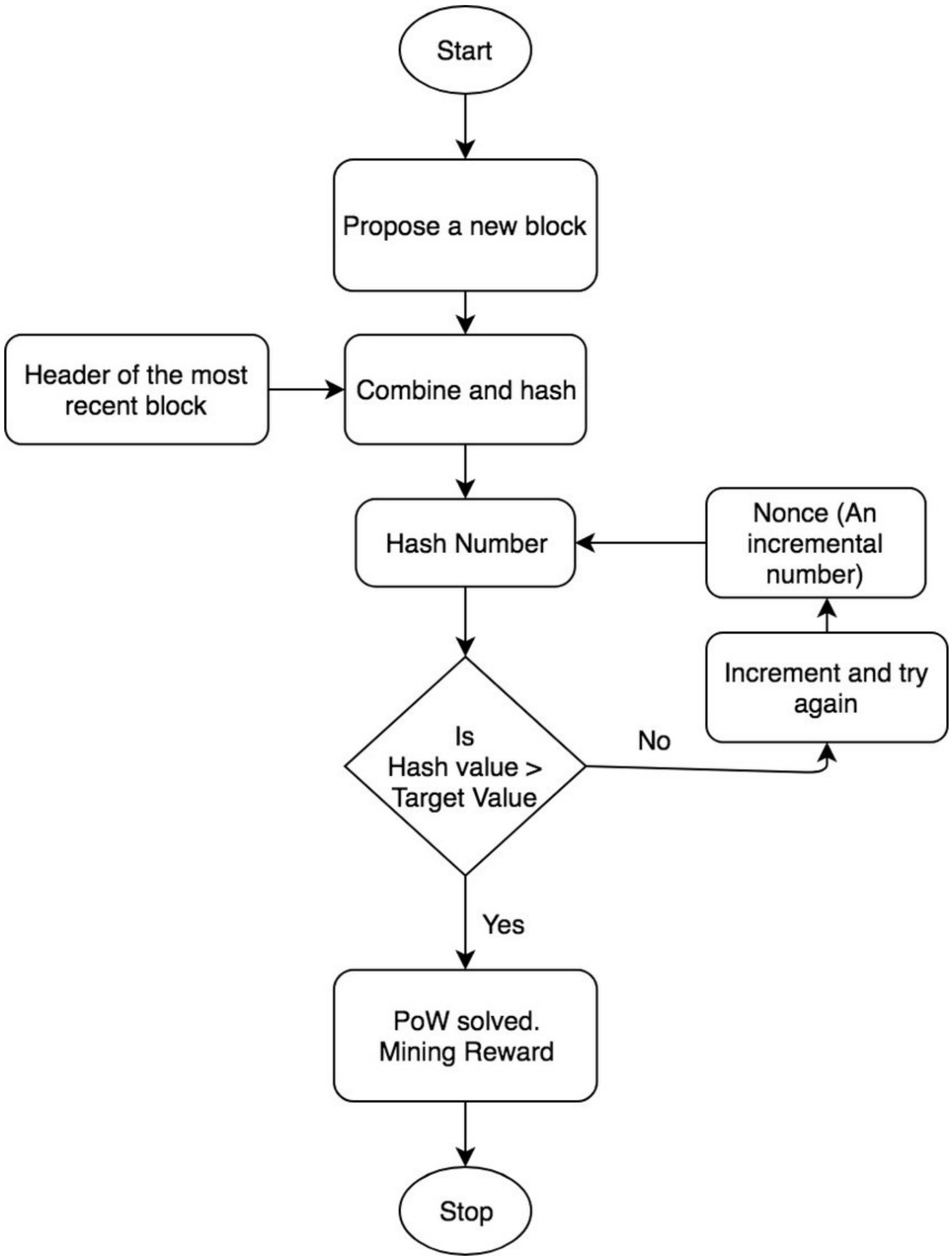
In PoS, validators are chosen to create new blocks based on the number of tokens they hold and are willing to "stake" as collateral. Unlike PoW, which relies on computational power, PoS relies on the economic stake in the network. [7]
Key properties and advantages of PoS:

- **Energy Efficiency**: PoS eliminates the need for intensive computational work, reducing energy consumption.
- **Security through Economic Stake**: Validators are required to lock up a portion of their tokens as collateral. If they act maliciously, they risk losing their staked tokens. This economic disincentive promotes honest behavior.
- **Scalability**: PoS can achieve faster block times and higher transaction throughput compared to PoW.
- **Decentralization**: PoS allows more participants to become validators, promoting decentralization and network security.

Challenges and considerations of PoS:

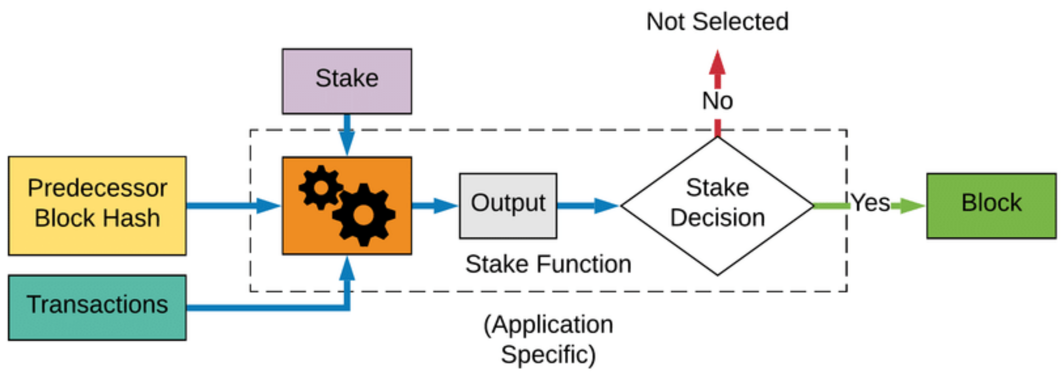
- **Wealth Concentration**: Validators with more tokens have a higher probability of creating new blocks, potentially leading to wealth concentration and centralization.

Figure 1.2 Proof of Work Chart [6]


Start
Propose a new block
Header of the most recent block
Combine and hash
Hash Number
Nonce (An incremental number)
Increment and try again
Is Hash value > Target Value
No
Yes
PoW solved. Mining Reward
Stop

- **Initial Distribution**: The initial distribution of tokens can impact the fairness and security of the PoS system.
- **Complex Implementation**: PoS mechanisms can be more complex to implement and require careful consideration of security and incentive structures.

Figure 1.3 Proof of Stake Flow [8]



1.5 Network Topology and Node Types

Blockchain networks operate across three primary classifications, each serving distinct purposes in the ecosystem. Public networks implement open participation models with economic incentives and require global consensus among participants. In contrast, private networks utilize permissioned access controls with defined validator sets and customized consensus mechanisms. Consortium networks bridge these approaches through hybrid permission models, enabling multi-stakeholder governance while maintaining custom consensus implementations.

The functionality of blockchain networks relies on different types of nodes working cooperatively. Full nodes form the network's backbone by performing complete blockchain validation, propagating transactions, and maintaining network state. Mining or validator nodes focus on block creation and validation while participating in consensus mechanisms to maintain network security. Light nodes enable resource-efficient participation through partial chain validation and transaction verification, making the network more accessible to resource-constrained participants.

1.6 Advanced Security Considerations

The security architecture of blockchain systems includes various layers of protection mechanisms and cryptographic safeguards. Understanding and implementing these security measures is essential for maintaining network integrity, defending against malicious actors, and ensuring the reliable operation of blockchain-based systems. These considerations must evolve to address emerging threats while maintaining the fundamental properties of decentralization and trustlessness.

Security in blockchain systems requires protection against various attack vectors.

Table 1.1 Attack Types and Mitigation Strategies

Attack Type	**Mitigation Strategy**
Network-Level	Implementation of 51% attack prevention, eclipse attack protection, and network partitioning resistance mechanisms
Protocol-Level	Deployment of transaction malleability prevention, replay attack protection, and double-spending prevention measures

The blockchain's security foundation rests on advanced cryptographic implementations. Signature schemes incorporate ECDSA, multi-signature protocols, and threshold signatures to ensure transaction authenticity. Zero-knowledge proofs enable privacy preservation through selective disclosure mechanisms while maintaining transaction confidentiality.

1.7 Future Development Trajectories

As blockchain technology evolves, scalability and interoperability become increasingly important for adoption and integration into various sectors. Understanding the future development trajectories of blockchain is essential for stakeholders looking to navigate this dynamic landscape effectively.

Blockchain technology's scalability advances along two main dimensions. Layer-1 scaling focuses on fundamental improvements to the protocol, such as optimizing consensus mechanisms, adjusting block parameters, and enhancing network efficiency. In contrast, Layer-2 solutions increase capacity through off-chain methods, including state channels, sidechains, and rollups. These solutions boost transaction throughput while maintaining security.

The future success of blockchain technology will depend on effective interoperability solutions. By developing cross-chain communication protocols, we can transfer assets and information through atomic swaps, bridge protocols, and cross-chain messaging systems. This progress is supported by ongoing standards development, which includes protocol standardization, interface specifications, and interoperability protocols.

Establishing these frameworks is crucial for creating a more integrated blockchain ecosystem. Such integration will allow different networks to interact with each other while preserving their unique features and ensuring robust security.

1.8 Conclusion

Blockchain technology's architectural principles and technical implementations represent a significant advancement in distributed systems design. Through cryptographic security, consensus mechanisms, and distributed networking, blockchain provides a foundation for trustless, decentralized applications. This chapter has established the theoretical and practical foundations for understanding more specialized implementations, particularly the Ethereum platform and smart contract systems, which will be examined in the next chapter. [9, 10, 6]

Chapter 2

Ethereum

Ethereum is a blockchain-based platform that extends Bitcoin's functionality by supporting decentralized applications and smart contracts. Since its launch in 2015 by Vitalik Buterin and a group of co-founders, Ethereum has become a fundamental building block in the blockchain ecosystem.

2.1 Ethereum Architecture

Ethereum's architecture consists of several key components that work together to ensure the platform's functionality and security.

Ethereum's primary innovation is the Ethereum Virtual Machine, which enables the execution of smart contracts on the blockchain. This functionality allows developers to create decentralized applications that can function without intermediaries.

The Ethereum Virtual Machine is a Turing-complete virtual machine that executes smart contracts on the Ethereum network, functioning as a global decentralized computer. It can execute code in a trustless environment, providing significant computational power and flexibility.

Key characteristics of the EVM:

- **Turing Completeness**: Can execute any computational task given enough resources.
- **Isolation**: Each contract runs in isolation, protecting the blockchain from malicious code.
- **Determinism**: Ensures that the same input will produce the same output in any node.

2.1.1 Ethereum Network and Nodes

Ethereum's network includes various nodes with specific roles and responsibilities. Full nodes store the entire blockchain, validate transactions, and maintain network integrity. Light nodes, on the other hand, keep only block headers and rely on full nodes for transaction validation. Miner or validator nodes are crucial in the consensus process, either by mining new blocks or validating transactions in PoS systems.

Table 2.1 Node Types and Roles

Node Type	**Role and Responsibility**
Full Nodes	Store the entire blockchain, validate transactions, and maintain network state.
Light Nodes	Store block headers and rely on full nodes for transaction validation.
Miner/Validator Nodes	Participate in consensus by mining new blocks or validating transactions in PoS systems.

2.2 Solidity

Solidity is the primary programming language for writing smart contracts on Ethereum, featuring a statically-typed syntax influenced by JavaScript, Python, and C++. [11]

A typical Solidity contract in Code Listing 2.1 consists of state variables, functions, and events. The example demonstrates a simple storage contract.

Code listing 2.1 Solidity Smart Contract

```
contract SimpleStorage {
    uint public storedData;
    event DataStored(uint data);

    function set(uint x) public {
        storedData = x;
        emit DataStored(x);
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

2.3 Smart Contracts

Smart contracts in Ethereum are self-executing contracts with the terms of the agreement directly written into code. They execute actions automatically when predetermined conditions are met.

2.3.1 Gas

Gas serves as the fundamental unit of computational measurement in the Ethereum network, acting as a mechanism for resource allocation and network security. Every operation executed on the EVM consumes a predetermined amount of gas.

The gas system fulfills several critical economic functions within the Ethereum ecosystem. It prevents spam attacks that could overwhelm the network, making computation costly. Network validators receive fair compensation for their work through gas fees, ensuring the network's continued operation and security. The gas mechanism creates a market-driven priority system where users can pay more for faster transaction processing. Additionally, it helps manage network congestion through dynamic pricing, where gas costs adjust based on network demand.

Developers have multiple strategies available to optimize gas consumption in their applications. Efficient smart contract design minimizes computational resources required for operations. Batch processing of transactions combines multiple operations, reducing overall gas costs. Strategic timing of transactions during periods of lower network activity can result in lower gas fees. Furthermore, implementing gas-efficient design patterns helps reduce the overall cost of deploying and interacting with smart contracts.

The gas mechanism is essential for maintaining the security, efficiency, and economic sustainability of the Ethereum network. It supports a balanced ecosystem where computational resources are allocated based on market principles. [12]

2.3.2 Memory and Storage in EVM

Understanding how the EVM handles memory and storage is crucial for efficient smart contract development. The EVM uses distinct areas for different types of data storage.

- **Storage:** Storage is where persistent state variables are kept. These variables are stored on the blockchain and are accessible across all nodes. Access to storage is expensive in terms of gas, which makes it crucial to minimize unnecessary storage operations.

- **Memory:** Memory refers to a temporary data storage area used during contract execution. Although it is cheaper than storage, data in memory does not persist between transactions. This area is used for intermediate computations and data that only need to last for the duration of a function call.

- **Stack:** The stack holds small local variables and function arguments for short-term use during execution. It is used for operations that require quick access in a limited scope, such as handling variables inside loops or temporary values during calculations.

Example of storage and memory usage:

■ **Code listing 2.2** Storage Usage

```
contract DataHandling {
    uint[] public storageArray;

    function addToArray(uint[] memory memoryArray) public {
        for (uint i = 0; i < memoryArray.length; i++) {
            storageArray.push(memoryArray[i]);
        }
    }
}
```

2.3.3 Smart Contract Functions and Modifiers

Smart contracts in Solidity include various functions, each tailored to execute specific operations. Constructors are special functions that initialize the contract and execute once it is deployed. They set up initial state values and perform initializations necessary for the contract. Fallback functions are unnamed and executed when the contract receives Ether without data or when a function call does not match any existing function signature. The receive function is used for the same purpose but is explicitly defined to handle plain Ether transfers.

Below is an example of different function types in a contract, shown in Code Listing 2.3.

■ **Code listing 2.3** Solidity Functions

```
contract Example {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    fallback() external payable {
        // Fallback function logic
    }

    receive() external payable {
        // Receive function logic
    }
}
```

Similar to other programming languages, functions in Solidity have distinct visibility modifiers that govern their accessibility and behavior.

Public functions can be accessed by any external user or contract, enabling interaction with the contract from outside sources. In contrast, private functions are restricted to the contract in which they are defined, ensuring internal usage only. In addition, there are **internal** functions, which are similar to private functions but can also be called by other contracts that inherit from the defining contract. **External** functions can only be called from other contracts and transactions, but not internally.

Solidity also categorizes functions based on their interaction with the state. **View functions** are read-only; they can access the state but cannot modify it. These functions are marked with the `view` keyword. **Pure functions** are entirely independent of the contract's state. They neither read from nor write to the state, making them deterministic and side-effect free. These functions are marked with the `pure` keyword. [13]

Modifiers in Solidity are used to change the behavior of functions, often used to manage access control. They are prefixed by the `modifier` keyword. For example, the `onlyOwner` modifier in the code snippet below ensures that only the contract owner can call a specific function:

■ **Code listing 2.4** Solidity Modifier

```
modifier onlyOwner() {
    require(msg.sender == owner, "Not authorized");
    _;
}

function set(uint x) public onlyOwner {
    storedData = x;
}
```

Documenting functions in Solidity is essential for clarity and collaboration. Solidity uses the NatSpec (Ethereum Natural Specification) format for function documentation. NatSpec provides a standardized way to document a function's purpose, parameters, return values, and possible errors. A full list of NatSpec tags is shown in Table 2.2, followed by an example of NatSpec documentation in Code Listing 2.5.

By documenting functions and following best practices in Solidity, developers can ensure that their smart contracts are maintainable, secure, and efficient. [14]

2.4 Decentralized Applications

Decentralized applications, commonly known as dApps, are a key innovation within the blockchain ecosystem, particularly on the Ethereum network. These applications leverage the unique capabilities of smart contracts to facilitate a wide range of decentralized activities.

Code listing 2.5 NatSpec Documentation Example

```
/**
 * @title A Basic Tree Simulator
 * @author John Doe
 * @notice This contract is for fundamental tree simulations.
 * @dev Implementations of functions do not produce side effects.
 * @custom:experimental This contract is considered experimental.
 */
contract BasicTree {
    /**
     * @notice Calculate the approximate age of a tree in years.
     * @dev Utilizes a basic increment algorithm.
     * @param rings Rings determined from tree core sample.
     * @return Age of tree in years, rounded up.
     * @return Name of the tree type.
     */
    function calculateAge(uint256 rings) external virtual pure
      returns (uint256, string memory) {
        return (rings + 1, "generic tree");
    }

    /**
     * @notice Provides an estimated count of leaves on this tree.
     * @dev Currently returns a static value.
     */
    function leafCount() external virtual pure returns(uint256) {
        return 50;
    }
}

contract CitrusTree is BasicTree {
    function calculateAge(uint256 rings) external override pure
      returns (uint256, string memory) {
        return (rings + 2, "Citrus Tree");
    }

    /**
     * Returns the leaf count for this specific type of tree.
     * @inheritdoc BasicTree
     */
    function leafCount() external override pure returns(uint256) {
        return 75;
    }
}
```

Table 2.2 NatSpec Tags [14]

Tag	Description	Applicable Contexts
`@title`	A title that should describe the contract/interface	contract, library, interface, struct, enum, enum values
`@author`	The name of the author	contract, library, interface, struct, enum, enum values
`@notice`	Explain to an end user what this does	contract, library, interface, function, public state variable, event, struct, enum, enum values, error
`@dev`	Explain to a developer any extra details	contract, library, interface, function, state variable, event, struct, enum, enum values, error
`@param`	Documents a parameter just like in Doxygen (must be followed by parameter name)	function, event, enum values, error
`@return`	Documents the return variables of a contract's function	function, enum, enum values, public state variable
`@inheritdoc`	Copies all missing tags from the base function (must be followed by the contract name)	function, enum, enum values, public state variable
`@custom:...`	Custom tag, semantics is application-defined	everywhere

A key characteristic of dApps is that they are open-source. Being open-source implies that the codebase of these applications is accessible to the public, allowing anyone to inspect, modify, and improve it. This transparency encourages a collaborative environment where developers from diverse backgrounds can contribute to the project, enhancing its robustness and security. Open-source projects are often subject to strict code review, which can help identify potential vulnerabilities.

Another characteristic of dApps is their operation on decentralized networks, unlike traditional applications that depend on centralized servers. Decentralization is achieved through the distribution of data and computational tasks across a network of nodes, which ensures that no single entity can take excessive control over the application. This approach not only enhances security against single points of failure but also aligns with the core principles of blockchain technology – transparency, immutability, and resistance to censorship.

Moreover, dApps utilize tokens that are integral to their operation and governance. These tokens serve multiple purposes within the ecosystem. Primarily, they act as a mechanism for incentivizing participation and contributions from users and developers alike. For instance, participants may earn tokens as rewards for providing computational power, contributing to the development, or facilitating transactions. Tokens also play a crucial role in maintaining and securing the network, as they can be used in consensus mechanisms like PoS, where holders validate transactions. Furthermore, they can be employed in decentralized governance models, enabling token holders to vote on proposals and decisions affecting the future directions of the dApps.

DApps on the Ethereum blockchain are distinguished by their open-source development, decentralized operational network, and use of tokens as incentives and governance tools. These characteristics reinforce security, transparency, and inclusivity, demonstrating the potential of blockchain technology in creating decentralized, resilient, and collaborative digital ecosystems. [15, 16]

2.5 Challenges and Future Directions

Despite its widespread adoption and success, Ethereum faces challenges and areas for improvement. Ongoing research and development efforts aim to address these challenges and enhance the platform's capabilities.

2.5.1 Upgradability

One of the main challenges in smart contract development is enabling upgradability. Unlike traditional software, smart contracts are immutable once deployed. However, several techniques can be employed to achieve upgradability.

The Proxy Pattern is a design approach that uses a proxy contract to delegate calls to a separate logic contract. This architecture allows the proxy to be updated to point to a new logic contract, facilitating upgrades without disrupting functionality.

The Eternal Storage Pattern separates storage and logic into distinct contracts. This design enables updates to the logic contract without changing the storage contract, providing greater flexibility.

Contract Libraries serve as repositories for reusable code, allowing for independent updates that enhance efficiency and maintainability in smart contract development.

2.5.2 Scalability

Ethereum's transaction throughput is limited, which can lead to congestion and high fees, especially during peak usage periods. Enhancing scalability is crucial for supporting the platform's growth and encouraging adoption.

One promising technique to tackle this issue is sharding. Sharding divides the blockchain network into smaller, more manageable segments called shards. Each shard can process its transactions and smart contracts, allowing for parallel processing. This parallelism can boost the network's throughput.

Layer 2 solutions have been developed alongside sharding to improve performance of Ethereum. These solutions work on the main blockchain and help reduce congestion by handling processing tasks off-chain. One well-known Layer 2 solution is state channels. State channels let participants conduct transactions off-chain, only recording the outcome on the blockchain.

Another approach is rollups, which group multiple transactions into a single package that can be processed on the main chain, increasing efficiency. Among rollup methods, Optimistic Rollups and ZK-Rollups stand out. Optimistic Rollups operate on the assumption that off-chain transactions are valid unless proven otherwise, verifying them only if there is a challenge. On the other hand, ZK-Rollups employ zero-knowledge proofs to ensure the correctness of off-chain transactions, which can enhance security and efficiency.

In summary, tackling Ethereum's scalability challenges involves sharding, which boosts throughput by parallelizing transaction processing, and various Layer 2 solutions like state channels and rollups, which reduce congestion by offloading transactions from the main chain. Each of these strategies plays a vital role in making Ethereum more efficient and capable of handling greater demand.

2.5.3 Security

Ensuring the security of Ethereum's network and smart contracts maintains user trust and ecosystem stability. As dApps grow in complexity and value, security measures are essential.

One primary tactic is smart contract auditing by security companies, which involves code reviews to identify vulnerabilities. These audits combine manual reviews with automated tools to catch potential weaknesses before deployment.

Formal verification techniques also enhance security by using mathematical methods to prove that a smart contract behaves as intended, ensuring the accuracy of critical contracts handling transactions or sensitive data.

Bug bounty programs strengthen security by motivating developers and researchers to find vulnerabilities and rewarding them for their findings. This community effort often uncovers subtle issues that might be overlooked.

Practices such as providing secure coding guidelines, regular updates, and decentralized governance enhance the security of the Ethereum environment.

A multifaceted approach involving smart contract auditing, formal verification, and community involvement is essential for the security of the Ethereum network and for fostering a trustworthy blockchain ecosystem.

2.5.4 Governance and Decentralization

Effective governance models ensure the sustainable development and evolution of Ethereum. On-chain governance allows stakeholders to vote on protocol upgrades directly on the blockchain. In contrast, off-chain governance relies on community consensus and off-chain discussions, followed by implementation through Ethereum Improvement Proposals.

Additionally, Decentralized Autonomous Organizations manage decision-making through smart contracts that enable stakeholders to vote on proposals and changes.

2.5.5 Interoperability

Establishing seamless connectivity among diverse blockchain networks encourages the substantial growth of the decentralized ecosystem. Ethereum must evolve to facilitate interactions with many other blockchain systems to unlock its potential and fully enhance adoption.

Cross-chain communication protocols lay the groundwork for exchanges and asset transfers among digital environments. These protocols, often known as bridge protocols, serve as conduits that enable the transfer of cryptocurrencies and digital assets between Ethereum and alternative blockchain platforms such as Binance Smart Chain or Solana.

By following widely accepted standards and protocols such as Ethereum Improvement Proposals and Inter-Blockchain Communication, these networks can ensure compatibility and foster an ecosystem that encourages seamless interactions. This approach enhances liquidity across digital assets and creates a rich landscape of blockchain environments. As a result, users and developers can take advantage of the unique features offered by each network.

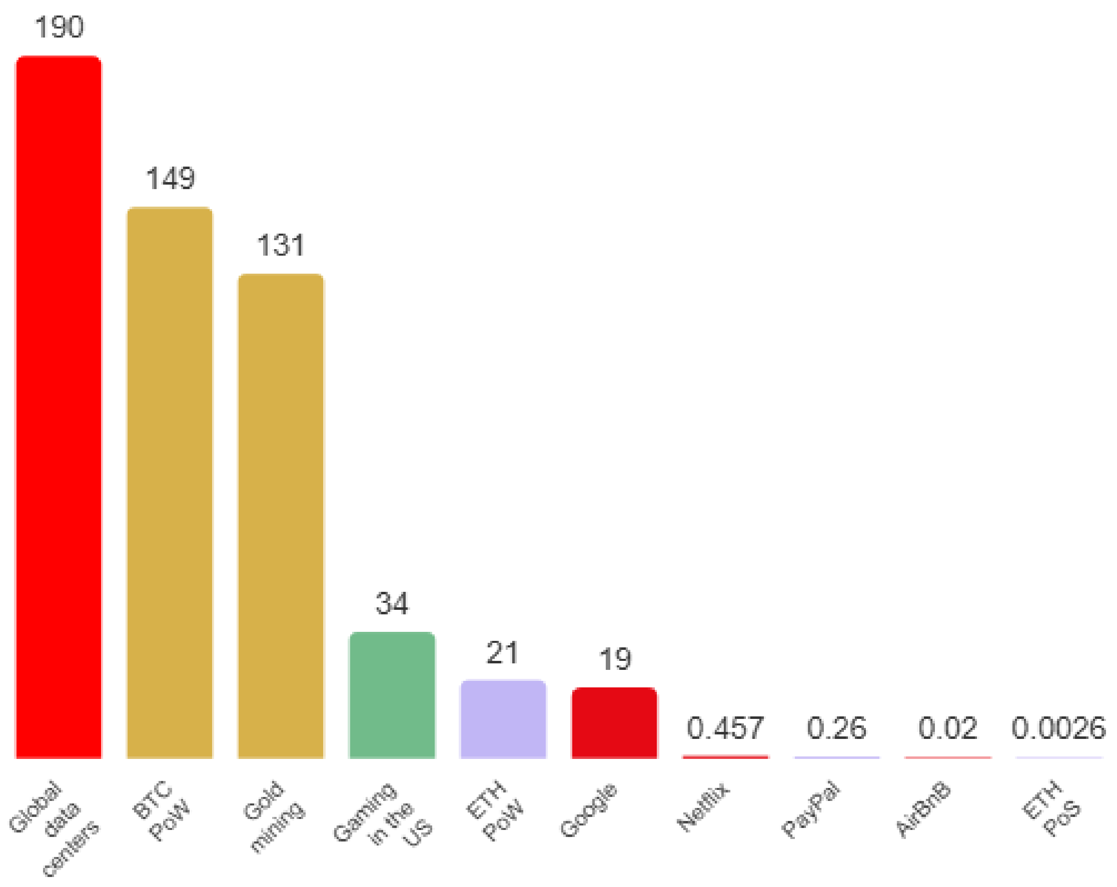
2.5.6 Sustainability

As the blockchain industry evolves and matures, the need for sustainable practices becomes important. Ethereum, a foundational component of this initiative, must prioritize its environmental impact and commit to a pathway of sustainable growth. The significant transition from PoW to PoS has greatly reduced energy consumption, paving the way for a greener future, see Figure 2.1. Furthermore, by collaborating with environmental organizations to offset its carbon footprint, Ethereum incorporates sustainability into its core operations. Embracing best practices in the design and deployment of smart contracts not only contributes to a healthier planet but also encourages responsible development that aligns with the values of a conscientious community. In this way, Ethereum can lead the blockchain revolution while advocating for environmental stewardship. [18, 19]

2.6 Conclusion

Ethereum has transformed the blockchain landscape with its support for smart contracts and decentralized applications. Its architecture, smart contracts, and ecosystem establish it as a foundational element of the blockchain environment. As Ethereum progresses, it will be essential to tackle challenges related to upgradability, scalability, security, governance, interoperability, and sustainability to ensure its future success and adoption. [20, 21]

Figure 2.1 Annual Energy Consumption in TWh/yr [17]



Chapter 3
Wake Framework

The Wake framework is a tool designed to aid in developing, testing, and static analysis of Solidity smart contracts. Developed by Ackee Blockchain, it offers integrated features that streamline the entire lifecycle of smart contract development, from initial coding and debugging to security auditing. Wake is particularly renowned for its capabilities in static code analysis, which helps identify potential vulnerabilities and bugs without executing the smart contract code. [1]

In this chapter, we will focus mainly on Wake's static code analysis capabilities, including its Intermediate Representation (IR) model and Abstract Syntax Tree (AST).

3.1 Overview of Wake

Wake provides a variety of functionalities designed to streamline the development of smart contracts. These functionalities include static code analysis, inspecting the code for vulnerabilities and bugs without actual execution; a testing framework for executing automated tests to ensure code correctness; development tools that enhance the coding experience with features such as syntax highlighting, code completion, and integrated debugging; and cross-chain testing, which enables the testing and deployment of contracts across different blockchain networks.

3.2 Static Analysis of Smart Contracts

Static code analysis in Wake involves examining the Solidity source code for potential issues without executing the code. This process includes several key steps. First, lexical analysis tokenizes the source code into a stream of tokens. Syntax analysis follows, constructing an Abstract Syntax Tree (AST) from these tokens. The semantic analysis then ensures that the code adheres to semantic rules and constructs the IR nodes. Finally, IR generation converts the AST into a more abstract representation suitable for analysis.

However, static analysis faces several challenges. One major challenge is balancing thoroughness with accuracy to avoid false positives (flagging non-issues) and false negatives (missing real vulnerabilities). The complexity and scalability of handling complex smart contracts also present obstacles. Another ongoing challenge is keeping analysis results up to date with evolving code, as smart contracts are frequently updated.

By addressing these challenges, Wake aims to provide an environment for the secure and efficient development of smart contracts.

3.3 Wake Framework Architecture

Wake utilizes a structured architecture that streamlines static code analysis through its dedicated components. At the core of this architecture are the Internal Representation (IR) model and the Abstract Syntax Tree (AST), which together allow for analysis and manipulation of Solidity smart contracts.

The IR model simplifies the structure of Solidity smart contracts, making them easier to analyze and manipulate. Preparing the IR involves several steps. First, the Solidity source code is compiled into bytecode using the Solidity compiler (`solc`), which generates an AST. This AST is then parsed and transformed into IR nodes, with each node representing different constructs within the smart contract. These IR nodes are serialized into a binary format for efficient storage and retrieval. To ensure security and prevent tampering, cryptographic keys are used to sign the hash of the serialized data. The IR nodes fall into various categories based on the constructs they represent, including function nodes (which encapsulate functions, including parameters, return types, and internal logic), variable nodes (which denote state variables, local variables, and their respective types), and control flow nodes (which represent control flow constructs such as loops and conditional statements). Each IR node also contains metadata that facilitates detailed inspection and manipulation, supporting analysis tasks.

The AST serves as a hierarchical tree representation of the syntactic structure of the Solidity code. Every node in the AST corresponds to a construct found in the source code. The generation of the AST involves parsing the source code into tokens and constructing a tree structure that encapsulates the code's syntax. Each node in the AST signifies a specific part of the syntax, such as expressions, statements, and declarations. The AST provides the groundwork for generating the IR, enabling organized code analysis.

By integrating these components, Wake offers a framework for the static analysis of smart contracts. This framework ensures inspection and manipulation while maintaining the integrity of the analysis process.

3.4 Working with IR

The Wake IR model builds on top of the AST produced by the Solidity compiler. It serves as a tree representation of the source code, holding additional information to simplify analysis.

The IR tree nodes can be divided into categories based on their functionalities:

- **Declarations:** Nodes that represent declarations of variables, functions, structs, etc.

- **Statements:** Nodes that control the execution flow (e.g., `if`, `for`, `while`) and nodes representing a single operation ending with a semicolon (e.g., assignments, function calls).
- **Expressions:** Nodes that typically have a value (e.g., literals, identifiers, function calls).
- **Type Names:** Nodes representing a type name (e.g., `uint`, `address`), usually used in a `VariableDeclaration`.
- **Meta:** Nodes typically used as helpers that do not belong to any of the above categories.

All expressions, type names, and a `VariableDeclaration` have type information attached to them. See the `wake.ir.types` API reference for more details. [22]

3.5 Structure of the IR Tree

The IR tree can have a very complex structure. However, certain rules make it easier to understand.

The root node of the IR tree is the `SourceUnit`. `FunctionDefinitions` and `ModifierDefinitions` hold statements, which may contain other statements and expressions.

Expressions can be used without a parental statement (i.e. outside of a function or modifier body) in specific cases:

- In an **InheritanceSpecifier** argument list, e.g. `contract A is B (1, 2)`.
- In a **StorageLayoutSpecifier** base slot expression, e.g. `contract C layout in (10 + 20)`.
- In a **ModifierInvocation** argument list, e.g. `function foo() public onlyOwner(1, 2)`.
- In a **VariableDeclaration** initial value, e.g. `uint a = 1;`.
- In an **ArrayTypeName** fixed length value, e.g. `uint[2] a;`.

Only a few nodes may reference other nodes, particularly declarations:

- **Identifier** as a simple name reference, e.g. `owner` referencing a variable declaration.
- **MemberAccess** as a member access reference, e.g. `owner.balance` referencing the global symbol `ADDRESS_BALANCE`.
- **IdentifierPathPart** as a helper structure used in `IdentifierPath` to describe a part of a path separated by dots, e.g. `Utils.IERC20`.
- **UserDefinedTypeName** as a reference to a user-defined type, e.g. `MyContract` in `new MyContract()`.

- **ExternalReference** as a helper structure describing a `YulIdentifier` referencing a Solidity `VariableDeclaration`, e.g. `assembly { mstore(0, owner) }`.

These rules and the organization of nodes help comprehend the IR tree's structure, aiding in the analysis and manipulation of the Solidity smart contracts.

3.5.1 Example IR Tree

The following example illustrates the complete IR tree for a simple Solidity code snippet:

Code listing 3.1 Solidity Code Snippet

```
library Math {
    function fib(uint n) public pure returns (uint) {
        if (n < 2) return n;
        return fib(n - 1) + fib(n - 2);
    }
}
```

In the IR tree for the above code, nodes of the same category are colored similarly. Dashed edges reference other nodes, highlighting connections within the IR structure, see Figure 3.1

3.6 Built-in Tools for Static Analysis

Wake includes built-in tools, such as the Control Flow Graph (CFG) and the Data Dependency Graph (DDG), to leverage the IR model for static analysis.

The CFG represents all possible execution paths within a smart contract. It helps detect issues such as unreachable code, infinite loops, and the improper use of control structures (misuse of loops and conditional statements affecting contract logic).

The DDG maps dependencies between data elements within a smart contract. This tool helps uncover uninitialized variables, improper data handling, and circular dependencies (cycles in data dependencies causing potential execution issues).

By utilizing these tools, Wake facilitates static analysis, helping developers identify and resolve issues in smart contracts.

Figure 3.1 IR Tree Example. [22]

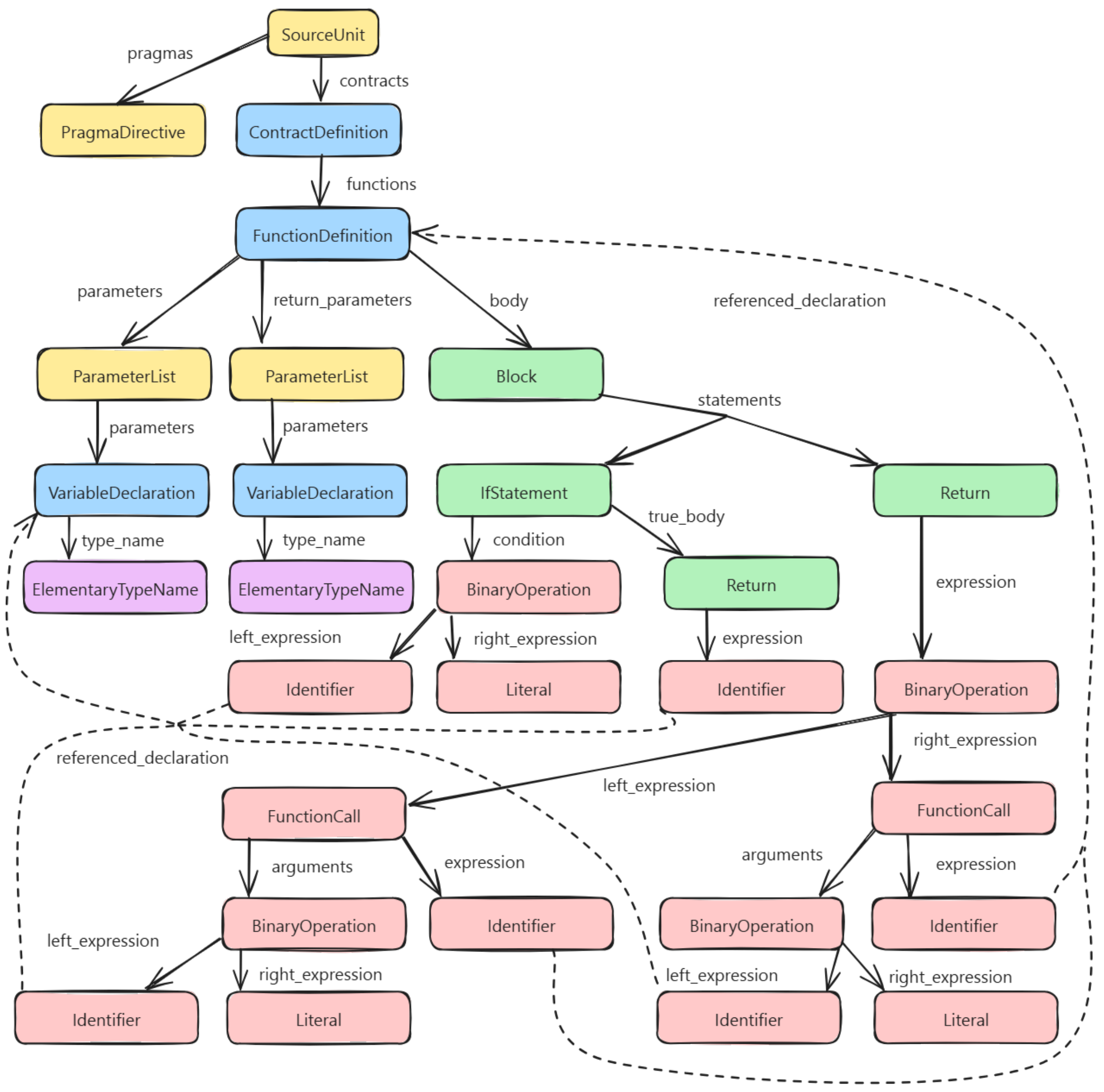
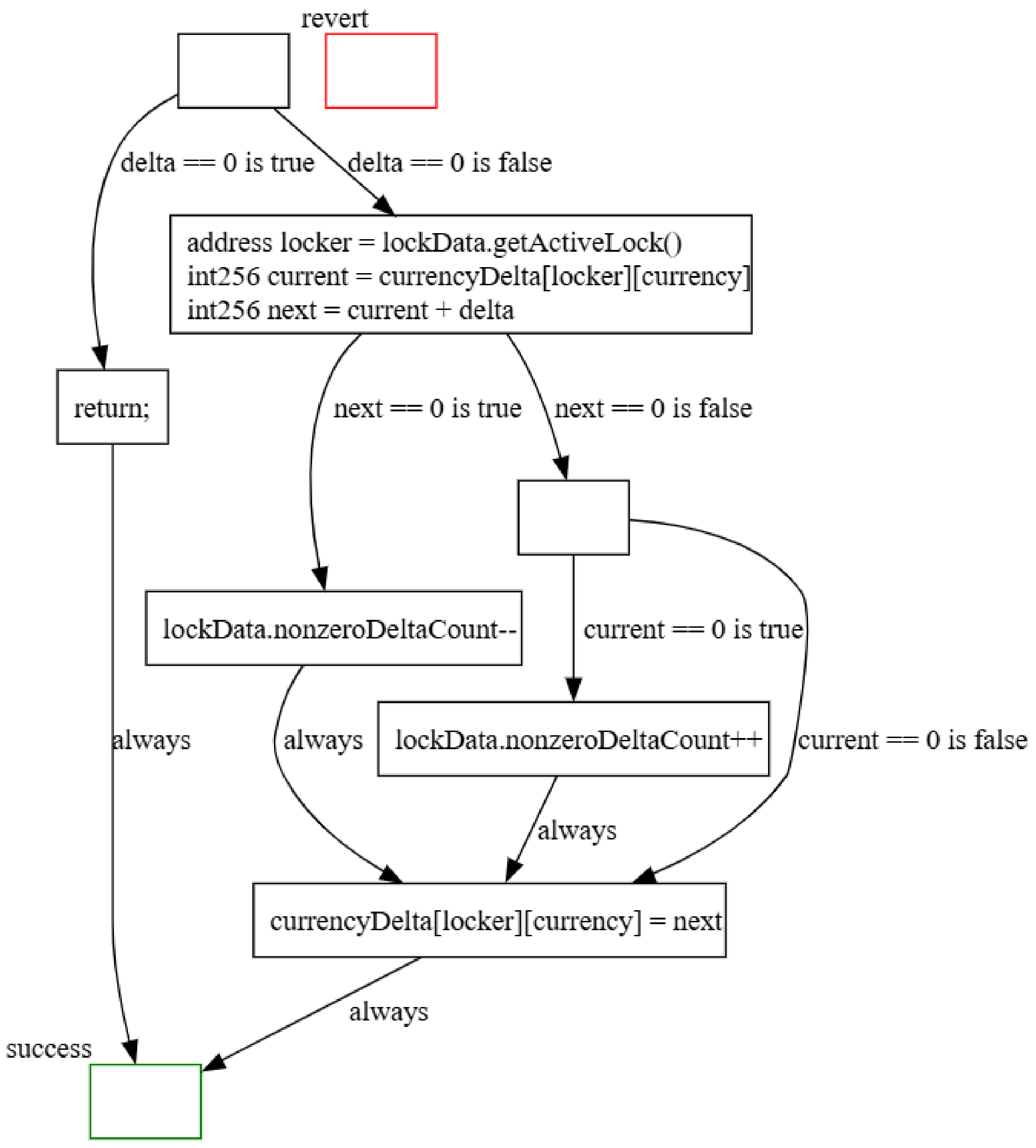


Figure 3.2 Control Flow Graph Example [23]


revert
delta == 0 is true
delta == 0 is false
address locker = lockData.getActiveLock()
int256 current = currencyDelta[locker][currency]
int256 next = current + delta
return;
next == 0 is true
next == 0 is false
lockData.nonzeroDeltaCount--
current == 0 is true
always
always
lockData.nonzeroDeltaCount++
current == 0 is false
always
currencyDelta[locker][currency] = next
always
success

3.7 Developing Custom Detectors

Wake's architecture allows developers to create and integrate custom detectors for specific analysis needs by subclassing from a base detector class and implementing specific analysis logic. For instance, a custom detector can be created by subclassing from the `BaseDetector` and overriding the `visit_` methods to traverse and analyze IR nodes.

■ **Code listing 3.2** Custom Detector

```python
class CustomDetector(BaseDetector):

    def visit_FunctionDefinition(self, node):
        # Custom analysis logic specific to function definitions
        pass

    def visit_VariableDeclaration(self, node):
        # Analysis logic for variable declarations
        pass
```

The Wake framework provides a set of `visit_` methods inherited from the `Visitor` class, which can be customized to analyze different types of IR nodes. For example, the `visit_FunctionDefinition` method is called when the detector visits a function definition node, allowing for the examination of function names, parameters, return types, and the function body. The `visit_VariableDeclaration` method analyzes variable declarations, detecting uninitialized variables, improper data types, and scope issues. Similarly, the `visit_IfStatement` method facilitates the analysis of if statement nodes for potential vulnerabilities like unchecked conditions, while the `visit_Assignment` method checks assignment statements for issues such as reassigning immutable variables or unintended data overwrites.

To exemplify how IR manipulation is crucial for practical analysis, consider the following detector that traverses variable declarations and reports any instances of uninitialized variables:

■ **Code listing 3.3** IR Traversal Example

```python
class UninitializedVariableDetector(BaseDetector):

    def visit_VariableDeclaration(self, node):
        if not node.initialized:
            self.report_issue(node, "Variable not initialized")

    def analyze(self, ir):
        for function in ir.functions:
            self.visit_FunctionDefinition(function)
```

In this example, the `UninitializedVariableDetector` class defines a method `visit_VariableDeclaration` that checks if a variable is initialized. If it is not, an issue is reported. During the analysis phase, the detector iterates over all functions in the IR, applying the custom analysis logic defined in `visit_FunctionDefinition` and `visit_VariableDeclaration`.

By leveraging these methods, developers can create detectors to perform detailed analyses of different code constructs, enhancing smart contract verification with the Wake framework.

3.8 Examples of Built-in Static Analyzers

Several pre-implemented detectors are available within Wake, specifically targeting common vulnerabilities in smart contracts. In the following overview, we will focus on a detailed examination of a selected subset of these detectors.

3.8.1 Reentrancy Detector

The reentrancy detector identifies potential reentrancy vulnerabilities in smart contracts, where malicious actors can repeatedly call a function before the previous execution finishes.

The detector flags public or external functions vulnerable to reentrancy due to lack of access control checks, untrusted external calls, and state-changing operations after external calls.

The primary class `ReentrancyDetector` in Code Listing 3.4 scans for reentrancy risks.

Code listing 3.4 ReentrancyDetector Class Core

```python
class ReentrancyDetector(Detector):

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

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

    def visit_member_access(self, node: ir.MemberAccess):
        # Implementation of ir.MemberAccess traversal

    def _check_reentrancy_in_function(
        self,
        function_definition: ir.FunctionDefinition,
        statement: ir.StatementAbc,
        address_source: ir.ExpressionAbc,
        child_modifies_state: Set[Tuple[ir.IrAbc,
          analysis.ModifiesStateFlag]],
        checked_statements: Set[ir.StatementAbc],
    ) -> List[Tuple[Detection, DetectorImpact, DetectorConfidence]]:
        # Implementation of the core function
```

Code listing 3.6 _check_reentrancy_in_function Snippet

```python
def _check_reentrancy_in_function(params...):
    # Imports
    ...
    source_address_declaration =
      find_low_level_call_source_address(address_source)
        is_safe = None
        if source_address_declaration is None:
            pass
        elif isinstance(source_address_declaration,
          ir.enums.GlobalSymbol):
            if source_address_declaration ==
               ir.enums.GlobalSymbol.THIS:
                is_safe = True
            elif source_address_declaration in {
                ir.enums.GlobalSymbol.MSG_SENDER,
                ir.enums.GlobalSymbol.TX_ORIGIN,
            }:
                is_safe = False
            else:
                is_safe = None
    ...
    # Further implementation
```

The `_check_reentrancy_in_function` method in Code Listing 3.6 traverses the contract's control flow to identify unsafe patterns. For the full version, see Appendix Code Listing A.1.

The detector flags the following vulnerable contract shown in Code Listing 3.5.

Code listing 3.5 Reentrancy Vulnerability Example

```solidity
contract Reentrancy {
    mapping(address => uint256) public balances;
    function withdraw(uint256 amount) public {
        require(balances[msg.sender] >= amount);
        // Vulnerable call
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success);
        balances[msg.sender] -= amount;
    }
}
```

In this contract, the call to `msg.sender.call` occurs before updating the balance, making it vulnerable to reentrancy attacks.

The reentrancy detector in the Wake framework identifies and flags potential reentrancy vulnerabilities in Solidity smart contracts, aiding developers in securing their code against such exploits.

3.8.2 tx.origin Detector

The `tx.origin` detector is used to identify potential vulnerabilities in smart contracts where the `tx.origin` field is used. Such usage can lead to phishing attacks and issues with ERC-4337 account abstraction.

The detector highlights two primary concerns: access controls based on `tx.origin` are vulnerable to phishing attacks, and use of `tx.origin` may prevent users using ERC-4337 account abstraction from interacting with a contract.

The class `TxOriginDetector` scans for unsafe usage of `tx.origin`, see Code Listing 3.7. The full version is available in Appendix Code Listing A.2.

In the contract example in Code Listing 3.8, an attacker can trick the owner into interacting with a malicious contract, which sets `tx.origin` to the owner's address and then triggers a withdrawal from the victim's contract.

Code listing 3.8 Phishing Attack Example

```
contract Victim {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    function withdraw() {
        require(tx.origin == owner);
        tx.origin.call{value: this.balance}("");
    }
}
```

In the contract shown in Code Listing 3.9, users utilizing account abstraction cannot deposit funds because `tx.origin` will not match the sender's address.

The `tx.origin` detector in the Wake framework helps identify and flag vulnerabilities in smart contracts related to phishing attacks and account abstraction issues by scanning for unsafe usage of `tx.origin`.

3.8.3 Unsafe delegatecall Detector

The `unsafe-delegatecall` detector identifies potential vulnerabilities in smart contracts where the `delegatecall` function is used to call untrusted contracts. Unsafe

Code listing 3.7 TxOriginDetector Class Snippet

```
class TxOriginDetector(Detector):

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

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

    def visit_member_access(self, node: ir.MemberAccess):
        if node.referenced_declaration !=
          ir.enums.GlobalSymbol.TX_ORIGIN:
            return
        if self._account_abstraction:
            self._detections.append(
                DetectorResult(
                    Detection(
                      node,
                      "Use of tx.origin may interfere with"
                        "ERC-4337 account abstraction",
                    ),
                    impact=DetectorImpact.WARNING,
                    confidence=DetectorConfidence.LOW,
                    uri=generate_detector_uri(
                      name = "tx-origin",
                      version =
                        self.extra["package_versions"]["eth-wake"],
                      anchor = "account-abstraction",
                    ),
                )
            )
    ...
    # Further implementation
```

`delegatecall` can lead to arbitrary code execution and storage modification.

The detector flags dangerous use of `delegatecall` unless the target address is trusted (e.g., `this`) or the call is protected by an access control modifier like `onlyOwner`.

The primary class `UnsafeDelegatecallDetector` in Code Listing 3.10 scans for unsafe usage of `delegatecall`. The `_check_delegatecall_in_function` traverses the function to check for unsafe `delegatecall` usage, see Code Listing 3.11. For the full versions, see Appendix Code Listings A.3 and A.4.

Code listing 3.9 Account Abstraction Example

```
contract Treasury {
    mapping(address => uint256) public deposits;

    function deposit() public payable {
        require(tx.origin == msg.sender);
        deposits[msg.sender] += msg.value;
    }
}
```

Code listing 3.10 UnsafeDelegatecallDetector Class Snippet

```
class UnsafeDelegatecallDetector(Detector):

    def init(self):
        self._detections = []

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

    def visit_member_access(self, node: ir.MemberAccess):
        t = node.type
        # Initial validation
        func = node.statement.declaration
        if not isinstance(func, ir.FunctionDefinition):
            return
        contract = func.parent
        if (
            not self._proxy
            and isinstance(contract, ir.ContractDefinition)
            and contract_is_proxy(contract)
        ):
            return
        ret = check_delegatecall_in_function(
            func, node.statement, node.expression, set()
        )
        if len(ret) == 0:
            return
        self._detections.append(
            DetectorResult(
                # Detection details
            )
        )
        ...
        # Further implementation
```

■ **Code listing 3.11** Delegatecall Check Function Snippet

```python
def check_delegatecall_in_function(
    function_definition: ir.FunctionDefinition,
    statement: ir.StatementAbc,
    address_source: ir.ExpressionAbc,
    checked_statements: Set[ir.StatementAbc],
) -> List[Tuple[Detection, DetectorConfidence]]:
    # Initial validation
    source_address_declaration =
      find_low_level_call_source_address(address_source)
    is_safe = None
    if source_address_declaration is None:
        pass
    elif isinstance(source_address_declaration, ir.enums.GlobalSymbol):
        if source_address_declaration == ir.enums.GlobalSymbol.THIS:
            is_safe = True
        elif source_address_declaration in {
            ir.enums.GlobalSymbol.MSG_SENDER,
            ir.enums.GlobalSymbol.TX_ORIGIN,
        }:
            is_safe = False
        else:
            is_safe = None
    ...
    # Further implementation
```

The detector flags the vulnerable contract, see Code Listing 3.12. In this contract, `computationLogic.delegatecall` is called without access control, allowing arbitrary code execution. The `unsafe-delegatecall` detector in the Wake framework helps identify and flag unsafe `delegatecall` usages in Solidity smart contracts, assisting developers in securing their code against such vulnerabilities.

3.9 Conclusion

The Wake framework is a powerful solution for the static analysis of Solidity smart contracts. It offers advanced features, extensibility, and a user-friendly interface for developing custom detectors. By addressing static analysis challenges and implementing tools, Wake enhances the security and reliability of Ethereum smart contracts. [24]

Code listing 3.12 Delegatecall Vulnerability Example

```
contract Storage {
    using SafeERC20 for IERC20;

    mapping(address => uint256) public balances;
    address public computationLogic;

    function setComputationLogic(address _computationLogic)
      external {
        computationLogic = _computationLogic;
    }

    function deposit(IERC20 token, uint256 amount) external {
        token.safeTransferFrom(msg.sender, address(this), amount);
        balances[msg.sender] += amount;
    }

    function recomputeRewards() external {
        computationLogic.delegatecall(
            abi.encodeWithSignature("recomputeRewards()")
        ); // Vulnerable call
    }
}
```

Chapter 4

Defining a Testing Set of Mainnet Contracts

This chapter explores the importance and methodology of defining a testing set of mainnet contracts for running vulnerability detectors. The ultimate goal is to evaluate the potential cost of exploits prevented by these detectors, ensuring the security and reliability of dApps deployed on the Ethereum blockchain.

4.1 Importance of Defining a Testing Set

Evaluating the security of smart contracts is crucial in the blockchain ecosystem, where vulnerabilities can lead to financial losses. Defining a representative testing set of mainnet contracts ensures that detectors are tested against real-world scenarios, providing realistic insights into their effectiveness. This process assesses the performance of vulnerability detectors in identifying exploits, validating their efficacy and reliability. Moreover, it allows for calculating potential financial losses that could have been incurred if vulnerabilities were not detected and mitigated, demonstrating the value of security measures.

4.2 Collecting Mainnet Contracts

A toolkit containing multiple scripts was developed to collect and work with a testing set of contracts. One of the scripts parses web pages on the DexScreener platform, extracting recently listed token trading pairs and allowing us to compile a relevant selection of mainnet contracts. This curated set of contracts is a foundation for testing detectors, ensuring they operate effectively in a real-world environment.

4.2.1 Application Workflow

The workflow of the collection process includes the following steps:

- **Parsing New Pairs:** The program parses the new-pairs page on DexScreener, gathering contract addresses of recently deployed contracts that meet specific filters.

- **Filtering Contracts:** Filters are applied to ensure that only contracts with specific characteristics are included in the testing set. This helps focus on contracts that are more likely to be targeted by attackers.
- **Calculating Total Locked Value:** After gathering the contract addresses, we run a script that calculates Total Value Locked in USD. This figure indicates the total worth of locked assets, serving as an indicator of the potential value at stake. It represents assets that could be at risk.
- **Downloading Source Codes:** Another script downloads the source codes of these contracts collected from the Etherscan platform, ensuring that the vulnerability detectors can analyze the contracts.

The script designed to calculate Total Value Locked (TVL) utilizes the DexScreener API, which supplies information on liquidity [25]. Liquidity represents the value held within smart contracts that enable trading and provide transaction support. We will obtain liquidity values specifically for the base tokens to evaluate overall risk exposure. It is important to note that only the TVL of the base tokens will be included in the final calculations, as potential vulnerabilities are unlikely to impact the values of other tokens.

The script for downloading source codes takes contract addresses gathered by the parsing script and integrates with the Etherscan API to fetch the source code of each contract. The downloaded source codes are then stored in a local repository for analysis.

4.3 Chosen Testing Set of Contracts

After implementing the script to collect and filter contracts, a result set was defined for evaluation purposes. The selection process involves filtering deployed smart contracts on the Ethereum mainnet using the criteria shown in Table 4.1. This ensures the inclusion of contracts that are recently deployed, have active engagement, and pose significant financial interest, making them potential targets for security threats.

Table 4.1 Filtering Criteria for Mainnet Contracts

Chain	Ethereum
Rank By	trendingScoreH6 (last 6 hours)
Order	Ascending
Minimum Liquidity	$10,000
Maximum Age	720 hours (30 days)

A subset of 100 contracts was chosen from the filtered list for practical evaluation. This number balances the need for a representative sample size with manageable complexity for detailed analysis. The addresses of these 100 selected contracts were collected and used to fetch the corresponding smart contract source codes and calculate their TVL.

The final set of 100 contracts was characterized by the following:

- **Liquidity:** The selected contracts all had a minimum liquidity of $10,000, ensuring they are significant enough to be of interest to potential attackers. This focus helps in understanding the impact of vulnerabilities in contracts that are actively traded and hold substantial value.

- **Active Engagement:** All chosen contracts exhibited significant user engagement, as indicated by their trending scores and liquidity metrics.
- **Recent Deployment:** With a maximum age of 30 days, the contracts were all very recent, ensuring the relevance of security evaluations to current deployment practices.

The sum of TVLs of all contracts: **$310,495,116.8**

4.4 Summary of the Selection Process

The parsing process narrowed down an array of mainnet contracts to a set for evaluation. This defined set includes real-world contracts that are actively used and exhibit significant activity and financial stakes, ensuring the evaluation's relevance.

By examining this selected collection of 100 contracts, we can evaluate the effectiveness of vulnerability detectors. It offers a deeper understanding of their practical relevance and potential financial benefits in preventing exploits, demonstrating their advantages in enhancing smart contract security.

4.5 Conclusion

Defining a testing set of mainnet contracts and evaluating them using vulnerability detectors is essential for ensuring the security of decentralized applications. The toolkit developed to gather and analyze these contracts provide a practical approach to understanding and mitigating potential risks. By evaluating the financial cost of potential exploits, we can illustrate the benefits of integrating advanced security measures into the smart contract development lifecycle.

Chapter 5

Evaluation

In this chapter, we will evaluate the effectiveness of the Wake framework by analyzing the results it produced, focusing on detections with high impact. We will gather the results into a table and then evaluate the potential cost of exploits based on these high-severity detections. The financial impact will be estimated by assessing the TVL of the affected contracts.

5.1 Running Detectors on the Testing Set

Once the source codes of the selected mainnet contracts are downloaded, they are subjected to a series of vulnerability detectors. Each detector analyzes the contracts for specific vulnerabilities, such as reentrancy, overflow/underflow, uninitialized variables, and others. A list of detectors is available on the documentation page [26].

The detectors classify security findings using five distinct impact levels: `INFO` $<$ `WARNING` $<$ `LOW` $<$ `MEDIUM` $<$ `HIGH`. These levels follow a strict ordering relationship.

Additionally, detections are assigned one of three confidence levels indicating the reliability of the finding: `LOW`, `MEDIUM`, and `HIGH`. The detection results are implemented through the `DetectorResult` class, which includes detection location and message, impact assessment, confidence rating, and an optional URI for detailed documentation. This classification system facilitates the prioritization of security findings based on severity and certainty of detection.

The execution process involves the following:

- **Wake Installation:** Installing the Wake framework and all required dependencies.
- **Input Preparation:** Arranging the downloaded source codes for compatibility with the Wake framework.
- **Analysis:** Running each detector against the source codes to identify potential vulnerabilities.
- **Result Compilation:** Compiling the results of the analysis, including detected vulnerabilities and their severities.

5.2 Automation of Detector Execution

Manual execution of detectors across multiple Solidity contracts presents several challenges. The process can be time-consuming, requiring individual detectors for each contract to be run separately. This makes the task inefficient and complicates the organization and comparison of results, as managing multiple outputs can be bulky. Additionally, there is a significant risk of human error when performing repetitive tasks, leading to inaccuracies and potentially critical oversights in the analysis. These challenges highlight the need for more streamlined and automated approaches in evaluating Solidity contracts.

The automation approach introduces several key benefits that enhance the detection workflow. First, it enables batch processing, allowing detectors to execute across multiple contracts or projects sequentially without the need for manual intervention. This streamlining saves time and reduces the potential for human error. The automation ensures a consistent environment, as all detectors operate under identical configuration parameters. This uniformity leads to more reliable and comparable results. The automated structured output organizes the detection results, facilitating easier analysis and comparison among the findings. Lastly, the process supports temporal analysis, which enables tracking vulnerabilities over time by maintaining an output history.

This approach improves the efficiency of the detection workflow. For this purpose, the implemented toolkit for contract parsing and downloading was extended by another script, running Wake commands and generating structured text output files that can be further processed for deeper analysis.

This automated approach enhances the efficiency of the Wake detector framework, transforming it from a tool-based process to a systematic security analysis methodology.

5.3 Analysis of Found Detections

The analysis begins by compiling all detections identified by the Wake framework. The primary focus is on detections categorized by their impact and confidence levels. This ensures a structured and prioritized assessment of potential vulnerabilities.

5.3.1 Results

This security analysis report presents the Wake framework's vulnerability assessment findings. The analysis identified 1120 potential security concerns across various severity levels. The vulnerabilities are categorized by their impact severity (HIGH, MEDIUM, LOW, WARNING, INFO), confidence level of detection, and specific vulnerability type. Notable findings include 32 high-impact reentrancy vulnerabilities, 109 unchecked return values across different severity levels, 35 unsafe ERC20 calls, and numerous code optimization opportunities identified through unused code elements, as shown in Table 5.1.

Figure 5.1 Severity Distribution

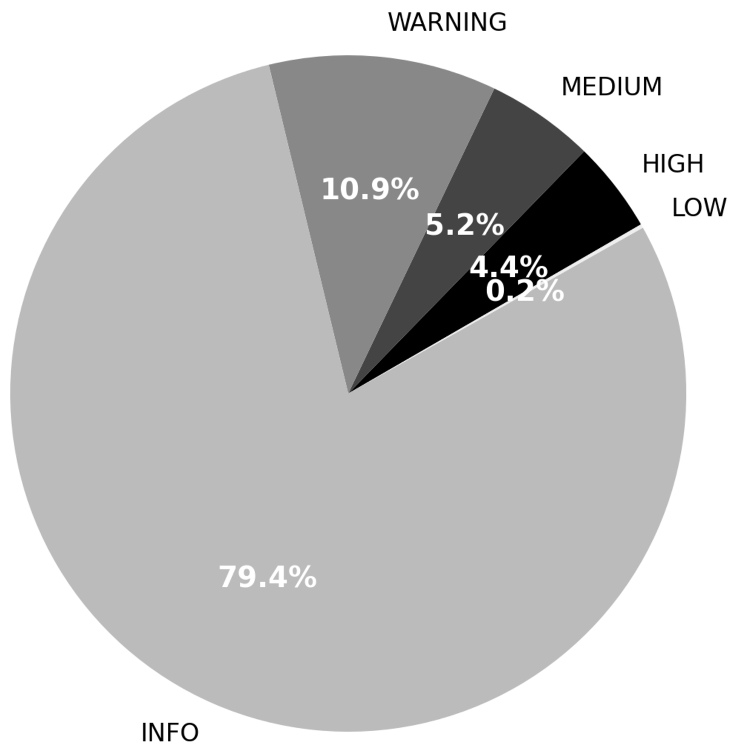
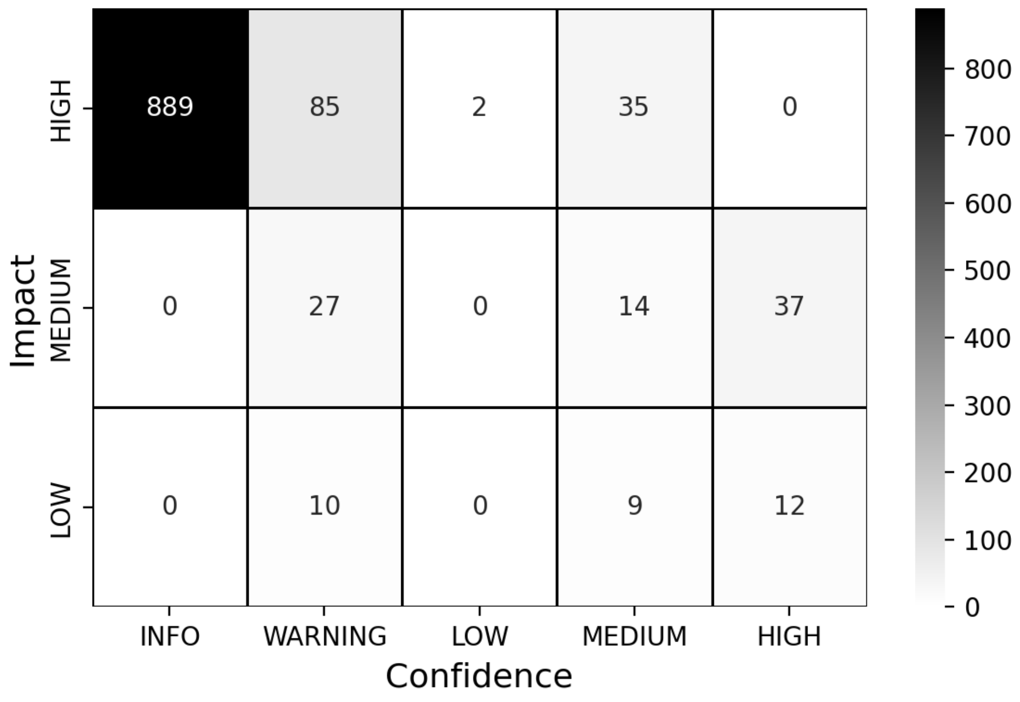


Figure 5.2 Heatmap of Foundings



Impact \ Confidence	INFO	WARNING	LOW	MEDIUM	HIGH
HIGH	889	85	2	35	0
MEDIUM	0	27	0	14	37
LOW	0	10	0	9	12

5.4 Evaluation of Potential Cost of Exploits

To evaluate the potential cost of exploits, we identify the affected contracts, focusing on those smart contracts that are impacted by high-severity detections. Once we have determined which contracts are at risk, we will assess the token values associated with these contracts. It involves retrieving the current token values or assets that may be compromised due to the identified vulnerabilities. Finally, we estimate the financial impact by calculating the potential loss for each vulnerability based on the assessed token values. This approach allows us to understand the financial implications of the detected issues.

Table 5.1 Global Vulnerability Statistics

Impact	**Confidence**	**Type**	**Count**
HIGH	MEDIUM	reentrancy	21
HIGH	MEDIUM	unchecked-return-value	15
HIGH	MEDIUM	unprotected-selfdestruct	1
HIGH	LOW	reentrancy	11
HIGH	LOW	calldata-tuple-reencoding-head-overflow-bug	1
MEDIUM	HIGH	unsafe-erc20-call	35
MEDIUM	MEDIUM	unchecked-return-value	10
MEDIUM	MEDIUM	unsafe-delegatecall	4
MEDIUM	LOW	tx-origin	9
LOW	HIGH	incorrect-interface	2
WARNING	HIGH	unchecked-return-value	84
WARNING	HIGH	complex-struct-getter	1
WARNING	MEDIUM	missing-return	27
WARNING	LOW	tx-origin	9
WARNING	LOW	reentrancy	1
INFO	HIGH	unused-function	833
INFO	HIGH	unused-contract	42
INFO	HIGH	unused-import	8
INFO	HIGH	unused-modifier	6
TOTAL			**1120**

5.4.1 Filtering High Impact and High Confidence

We filter detections classified as high impact to evaluate potential financial consequences. This approach ensures our attention is concentrated on the most critical vulnerabilities with the highest potential for financial outcomes.

Considering the absence of vulnerabilities categorized with high impact and high confidence, our analysis will focus on those with medium confidence. Detections with low confidence are removed from this evaluation due to a higher likelihood of false positives, which require manual verification for confirmation.

The Table 5.2 presents only the affected contracts and their corresponding high-impact, medium-confidence detections.

Focusing on high-severity detections is essential for several reasons. First, high-impact vulnerabilities pose critical risks, as they can lead to security threats that may

Table 5.2 Summary of High Impact Vulnerabilities

Contract	Impact	Confidence	Count	Type
Ecotrader	HIGH	MEDIUM	10	reentrancy
BiorBank	HIGH	MEDIUM	10	reentrancy
WA7A5	HIGH	MEDIUM	2	unchecked-return-value
POWER	HIGH	MEDIUM	2	unchecked-return-value
DecentralizedEURO	HIGH	MEDIUM	2	unchecked-return-value
XION	HIGH	MEDIUM	1	unprotected-selfdestruct
Ghibli	HIGH	MEDIUM	1	unchecked-return-value
MIDAS	HIGH	MEDIUM	1	unchecked-return-value
DOPE	HIGH	MEDIUM	1	unchecked-return-value
mipramilekibro	HIGH	MEDIUM	1	unchecked-return-value
Koko-chan	HIGH	MEDIUM	1	unchecked-return-value
SpaceChain	HIGH	MEDIUM	1	unchecked-return-value
LegalXToken	HIGH	MEDIUM	1	unchecked-return-value
LegalXToken	HIGH	MEDIUM	1	reentrancy
COCORO	HIGH	MEDIUM	1	unchecked-return-value
BitBonds	HIGH	MEDIUM	1	unchecked-return-value

result in financial losses or even a complete compromise of contracts. Since security teams often operate with limited resources, prioritizing critical vulnerabilities is vital for managing and mitigating risk. Furthermore, addressing high-impact issues results in a greater reduction in risk per unit effort compared to lower-impact issues. The potential for considerable reputational damage from exploiting high-severity vulnerabilities underlines the urgency of their timely resolution, as safeguarding an organization's reputation is crucial in today's digital landscape.

5.4.2 Calculating Value at Risk

After identifying contracts affected by high-impact vulnerabilities, it is necessary to calculate the TVL of the base tokens at risk.

The results are compiled in the Table 5.3, which details the TVL of each affected contract. It is important to note that some values do not exceed the applied filter of $10,000 in minimal liquidity, as only base token values held by affected smart contracts are considered for evaluation.

Table 5.3 Summary of High Impact Vulnerabilities

Contract	**Type**	**TVL**
Ecotrader	reentrancy	$34,406.17
BiorBank	reentrancy	$36,474.65
WA7A5	unchecked-return-value	$1,981,979.53
Power Play	unchecked-return-value	$3,458.93
DecentralizedEURO	unchecked-return-value	$105,464.80
XION	unprotected-selfdestruct	$27,305.65
Ghibli	unchecked-return-value	$8,487.17
MIDAS	unchecked-return-value	$29,239.24
DOPE	unchecked-return-value	$41,072.92
mipramilekibro	unchecked-veturn-value	$33,771.55
Koko-chan	unchecked-return-value	$10,961.06
SpaceChain	unchecked-return-value	$85,452.04
LegalXToken	unchecked-return-value, reentrancy	$41,481.91
COCORO	unchecked-return-value	$257,240.92
BitBonds	unchecked-return-value	$4,690.24

5.5 Detailed Examination

5.5.1 Reentrancy

1. **Contract**: Ecotrader
   Line Numbers: 159, 168, 170, 176, 186, 187, 384, 389, 422, 434
   Description: The contract contains multiple high-severity reentrancy vulnerabilities where external calls are made before state changes are finalized, allowing attackers to re-enter the contract and manipulate its state. Most critical reentrancy points occur in the launch(), cancel(), and processFees() functions, where calls to external contracts like Uniswap router and token transfers are performed. These vulnerabilities could lead to the theft of funds, manipulation of token balances, or circumvention of liquidity-locking mechanisms. The MEDIUM confidence level indicates strong evidence of exploitability based on code patterns, though specific exploitation paths might depend on contract interactions. If exploited, these vulnerabilities could allow attackers to drain contract funds, manipulate token supply, or interfere with essential protocol operations.
   Estimated Financial Impact: $34,406.17

2. **Contract**: BiorBank
   Line Numbers: 157, 166, 168, 174, 184, 185, 385, 390, 423, 435
   Description: Similar behavior and functions as in the Ecotrader contract described above.
   Estimated Financial Impact: $36,474.65

3. **Contract**: LegalXToken
   Line Numbers: 2523

Description: This vulnerability is a classic reentrancy attack in the function _withdrawDividendOfUser, where the contract sends ETH to a user address before updating its state. When the ETH transfer is made using the user.callvalue: _withdrawableDividend(""), a malicious recipient contract can execute code and call back into the vulnerable contract's functions. This callback can happen before the original function completes, allowing the attacker to repeatedly withdraw funds that should no longer be available. The vulnerability is accessible through the public withdrawDividend() function, making it directly exploitable by external users who could drain funds from the contract.
Estimated Financial Impact: $41,481.91

5.5.2 Unchecked Return Value

1. **Contract**: WA7A5
   Line Numbers: 35, 51
   Description: The wA7A5 contract contains high-severity vulnerabilities related to unchecked return values and unsafe ERC-20 calls in the wrap() and unwrap() functions. The contract calls functions A7A5.transferFrom() and A7A5.transfer() without checking their return values, potentially leading to silent failures where tokens are not transferred, but the contract proceeds as if they were. This issue is dangerous because the contract mints new tokens before confirming the underlying assets have been received, creating a potential attack vector that could allow malicious users to obtain wrapped tokens without providing the required collateral. Some ERC-20 tokens do not follow the standard implementation and may return false instead of reverting on failure, allowing attackers to exploit this contract and inflate the wrapped token supply without proper backing. If exploited, these vulnerabilities could lead to the wA7A5 token becoming undercollateralized, causing significant financial loss to users and collapsing the entire wrapped token system.
   Estimated Financial Impact: $1,981,979.53

2. **Contract**: Power Play
   Line Numbers: 1971, 1983
   Description: The POWER contract contains high-severity issues related to unchecked return values from ERC-20 token operations in its liquidity provision functionality. When transferring WETH tokens to the liquidity pair in the addLp() function, the contract fails to verify if these transfers succeeded, potentially causing silent failures where liquidity appears to be added but tokens are not transferred. Some ERC-20 tokens (including certain implementations of WETH) return false on failed transfers rather than reverting, making this vulnerability exploitable if the underlying token behaves this way. This issue is dangerous during the liquidity provisioning stage, where tokens and ETH are being contributed, as it could result in mismatched reserves in the liquidity pool and allow manipulation of token prices. If exploited, this vulnerability could lead to incorrect liquidity creation, broken token economics, or sophisticated price manipulation attacks against the token and its users.

Estimated Financial Impact: $3,458.93

3. **Contract**: DecentralizedEURO
Line Numbers: 328, 399
Description: The DecentralizedEURO system contains high-severity vulnerabilities where ERC-20 token transfer return values are not checked in the Equity contract, potentially leading to silent failures that could corrupt the financial state of the system. The _invest() function calls dEURO.transferFrom() without verifying the transfer was successful, which could allow users to receive equity shares without transferring any tokens. Similarly, in the redemption process, dEURO.transfer() is called without checking its success, which could result in burned equity tokens without the corresponding dEURO tokens being transferred to users. These issues are hazardous since the protocol deals with financial assets, and silent transfer failures could lead to fund misallocations, incorrect accounting of reserves, or potential economic attacks where malicious users exploit discrepancies in the system's actual vs. recorded token balances. If exploited, these vulnerabilities could damage the protocol's reliability, leading to a complete breakdown of trust in the stablecoin ecosystem.
Estimated Financial Impact: $105,464.80

4. **Contract**: Ghibli
Line Numbers: 1101
Description: This vulnerability occurs when the contract calls IERC20 (_tokenAddr).transfer() but fails to check the return value that indicates whether the transfer was successful. Many ERC20 tokens return false instead of reverting when a transfer fails, which means the contract will continue executing as if the transfer succeeded even when it failed. This can lead to serious accounting errors where the contract thinks funds were moved when they were not, allowing users to perform actions they should not be able to do after a failed transfer. The vulnerability is dangerous because it can silently break the contract's core financial logic, creating discrepancies between the contract's understanding of token balances and the actual token distribution.
Estimated Financial Impact: $8,487.17

5. **Contract**: MIDAS
Line Numbers: 609
Description: This vulnerability occurs in the token distribution logic, where the contract updates accounting records without verifying if the token transfer was successful. The code first increases the totalDistributed counter and then calls IERC20(PAXG).transfer() without checking its return value, finally updating shareholderClaims timestamp. If the PAXG token returns false on transfer failure rather than reverting, the contract will continue execution, creating a dangerous accounting discrepancy where distribution records indicate tokens were sent when they were not. This is severe because the contract maintains state before and after the unchecked transfer, allowing shareholders to manipulate distribution records without actual to-

ken movement occurring, effectively breaking the entire dividend distribution system.
Estimated Financial Impact: $29,239.24

6. **Contract**: DOPE
Line Numbers: 305
Description: The DOPE contract contains high-severity unchecked return value vulnerabilities in its openTrading function that initializes the token's trading functionality. The contract fails to verify the success of critical ERC-20 operations, particularly when transferring tokens to itself and when approving the Uniswap pair contract to spend tokens. These silent failures could lead to a corrupted contract state where trading appears to be enabled, but underlying token transfers or approvals have failed. Since this occurs during the initial trading setup and liquidity provision process, exploitation could result in improper initialization of the trading pair, allowing the contract owner to extract funds without establishing proper liquidity. If exploited, these vulnerabilities could result in financial loss to early investors or manipulation of the initial token pricing mechanism.
Estimated Financial Impact: $41,072.92

7. **Contract**: mipramilekibro
Line Numbers: 307
Description: The mipramilekibro token contains unchecked return value vulnerabilities in its openTrading function, particularly with token transfers and approvals to the Uniswap pair. When initializing trading, the contract does not verify if critical operations like token transfers and LP token approvals succeeded before enabling trading. This could lead to an inconsistent contract state where trading appears enabled, but underlying operations have failed. If exploited during launch, these vulnerabilities could disrupt proper liquidity pool initialization or allow premature trading while the contract is unstable.
Estimated Financial Impact: $33,771.55

8. **Contract**: Koko-chan
Line Numbers: 331
Description: This vulnerability appears in the rescueERC20 function, which is designed to recover any ERC20 tokens held by the contract, but fails to verify if the token transfer operation succeeds. The function calculates a percentage of the contract's token balance to withdraw and calls IERC20(_address).transfer() without checking its return value, meaning that if the transfer fails silently (returning false instead of reverting), the function will complete execution as if the rescue was successful. This vulnerability is concerning because it occurs in a privileged emergency recovery function that handles arbitrary ERC20 tokens, potentially leading the contract administrator to believe tokens were successfully rescued when they remain trapped in the contract. Since the function provides no feedback about transfer failures, the tax wallet operator might never realize that the rescue operation failed, resulting in the permanent loss of supposedly "rescued" tokens.

Estimated Financial Impact: $10,961.06

9. **Contract**: SpaceChain
Line Numbers: 1707
Description: The SpaceChain contract contains a high-severity unchecked return value vulnerability in the TokenUpgrader's withdraw function, transferring tokens without verifying if the operation succeeded. When a user requests to withdraw their tokens from the upgrader contract, the function calls token.transfer() but fails to check its return value, potentially allowing silent failures where users believe tokens were sent when the transfer failed. This vulnerability is particularly concerning because it affects the token migration pathway, where users migrate from V1 to V2 tokens through the upgrader contract. If token transfers silently fail during migration, users could lose funds, believe they have properly migrated their tokens when they have not, or the contract state could become inconsistent with actual token balances. Additionally, a lower confidence reentrancy risk in the token migration function could be exploited if the V1 token has malicious callbacks.
Estimated Financial Impact: $85,452.04

10. **Contract**: LegalXToken
Line Numbers: 2523, 2535
Description: The LegalXToken contract contains a high-severity unchecked return value vulnerability in its dividend distribution mechanism, where token transfers are executed without verifying their success. In the function _withdrawDividendOfUser, the contract calls IERC20(RewardToken).transfer() but does not check if the transfer succeeded, potentially allowing silent failures where dividends appear to be paid but tokens are not transferred. This vulnerability is dangerous in a dividend-paying token since it affects the core economic function of the contract, potentially leading to accounting inconsistencies between claimed dividends and actual token transfers. If exploited, users could have their dividend rewards recorded as paid in the contract's state while the actual tokens remain in the contract, essentially losing their rightful rewards. Additionally, the contract has tx.origin usage concerns could interfere with smart contract wallet functionality and other unchecked return values throughout the dividend-claiming process.
Estimated Financial Impact: $41,481.91

11. **Contract**: COCORO
Line Numbers: 603
Description: The COCORO token contract contains critical unchecked return value vulnerabilities in its openTrading function that could compromise the initial token setup process. The function fails to verify the success of key operations, including transferring 98% of tokens to the contract itself and approving the Uniswap router to spend LP tokens. If these operations silently fail, the contract will proceed with liquidity provision and trading enablement even though the underlying transfers or approvals have not been completed successfully. This vulnerability is dangerous

during the critical one-time trading initialization process, potentially resulting in a corrupted trading pair or allowing the owner to extract ETH without properly contributing tokens to the liquidity pool. If exploited, early investors could experience financial losses due to inadequate liquidity provision or encounter a liquidity pool with imbalanced reserves that enable price manipulation.
Estimated Financial Impact: $257,240.92

12. **Contract**: BitBonds
Line Numbers: 661
Description: This vulnerability combines an unchecked ERC20 token transfer with a subsequent ETH transfer, creating a dangerous scenario where funds could be partially lost. The function first attempts to transfer all tokens of a specified address to mkt without checking if the operation succeeded, then immediately proceeds to transfer all ETH balances regardless of the token transfer's outcome. Since some popular ERC20 tokens return false on failure rather than reverting, the token transfer could silently fail while the ETH transfer completes successfully. This partial execution creates an inconsistent state where the caller believes both transfers succeeded when only the ETH was moved, potentially causing accounting errors or lost tokens. The vulnerability is concerning because it appears in a privileged sweeping function intended to recover assets from the contract, but may result in tokens becoming permanently trapped.
Estimated Financial Impact: $4,690.24

5.5.3 Unprotected self-destruct

1. **Contract**: XION
Line Numbers: 1461
Description: This vulnerability involves an unprotected selfdestruct function that allows anyone to destroy the contract and transfer all its ETH to an arbitrary address. The destroy function is marked as external with only a noReenter modifier, which prevents reentrancy attacks but lacks access control mechanisms like onlyOwner to restrict who can call it. Any external actor can call this function at any time, causing permanent destruction of the contract, loss of all stored data, and transfer of the contract's entire ETH balance to an address of the attacker's choosing. This represents a critical security vulnerability as it enables complete and irreversible contract destruction by unauthorized parties, potentially resulting in permanent loss of funds and functionality.
Estimated Financial Impact: $27,305.65

5.6 Summary

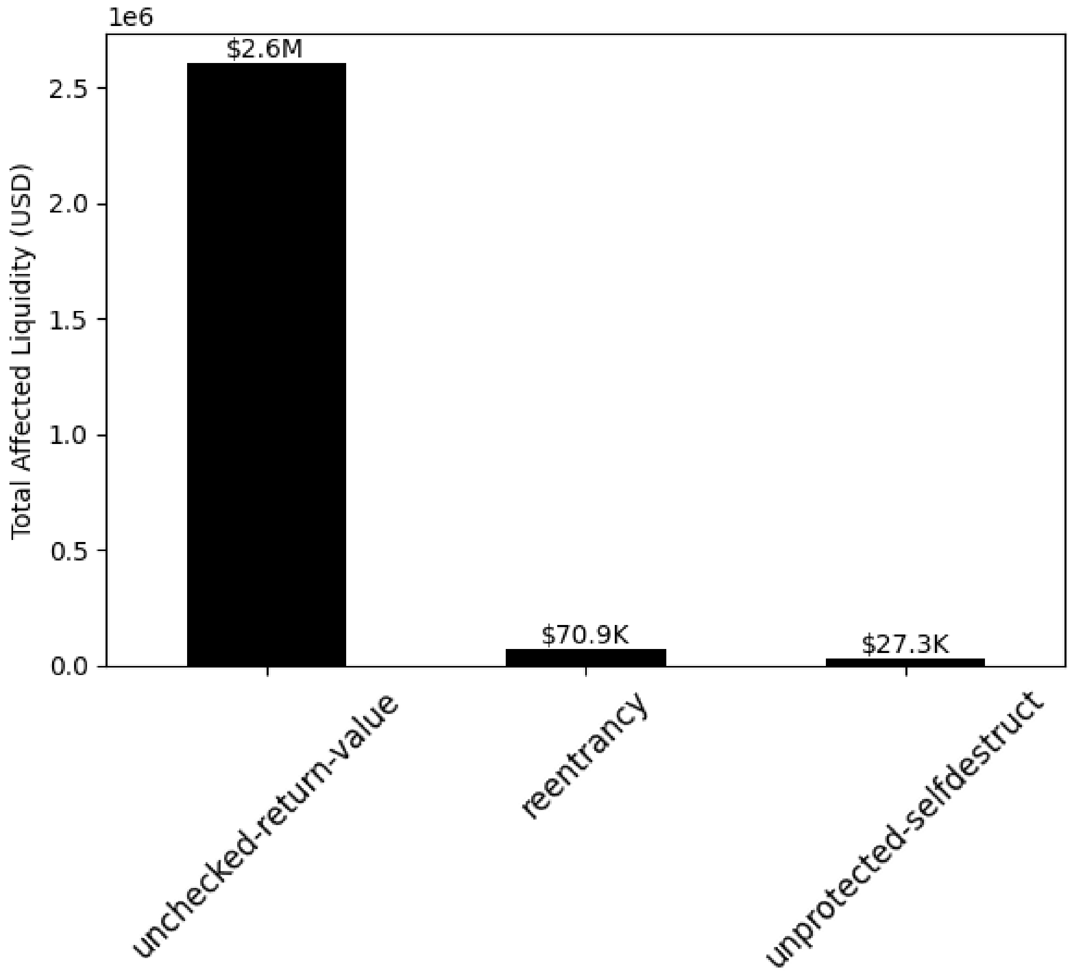


Figure 5.3 Total Liquidity Affected by Vulnerability Type

Total Cost of Potential Exploits: **$2,701,486.78**

The evaluation process demonstrates the importance of prioritizing high-impact detections when assessing security risks in smart contracts. Focusing on these critical vulnerabilities allows one to estimate the financial consequences and take necessary actions to mitigate associated risks.

The tables created during the analysis serve as a resource for prioritizing security efforts and ensuring that attention is directed toward the most pressing issues. This chapter outlined the steps in detecting, analyzing, and evaluating security issues within smart contracts, demonstrating the application of the Wake framework.

Chapter 6

Implementation of the New Detector

This chapter details the development process of the `documentation-diff` detector. This detector aims to identify inconsistencies between a function's documentation and its implementation using LLM for evaluation.

6.1 Motivation

Maintaining accurate and up-to-date documentation is crucial for code reliability and maintainability in modern software development. Misalignment between documentation, such as docstrings, and the actual code can lead to misunderstandings, misusages, and potentially faulty behavior. The detector `documentation-diff` helps address this issue by identifying inconsistencies between the documentation and the implementation of functions within smart contracts.

6.2 Design and Implementation

The `documentation-diff` detector is designed to traverse the IR of smart contracts using Wake's visitor pattern. It collects function definitions and their corresponding NatSpec documentation, and then uses LLM to evaluate consistency between the two.

6.2.1 Intermediate Representation Model

The detector operates on Wake's IR, concentrating on two crucial types of nodes. The first type is the **ir.ContractDefinition**, which encapsulates the semantics at the contract level. This includes elements such as state variables, modifiers, and function declarations, all providing the structural context necessary for analysis. The second type is the **ir.FunctionDefinition**, which represents individual implementations of functions. It includes associated metadata, documentation strings, and relationships with modifiers, allowing for a detailed understanding of each function's role within the contract.

6.2.2 Detector Class and Initialization

The `DocumentationDiffDetector` class in Code Listing 6.1 extends the `Detector` base class provided by the Wake framework. It initializes the internal detection list, sets up function caches, and retrieves API keys necessary for calls to AI for evaluation.

Code listing 6.1 Detection Initialization

```
class DocumentationDiffDetector(Detector):

    NAME = "documentation-diff"
    IMPACT = DetectorImpact.WARNING
    CONFIDENCE = DetectorConfidence.MEDIUM

    def __init__(self) -> None:
        self._detections: List[DetectorResult] = []
        self._llm_cache: Dict[str, str] = {}
        self._contract_functions: Dict[ir.ContractDefinition,
          List[Tuple[ir.FunctionDefinition, Dict]]] = {}
        self._node_to_contract: Dict[ir.FunctionDefinition,
          ir.ContractDefinition] = {}
        api_key = self.config.api_keys.get("anthropic_api_key")
        ...
        # Further implementation
```

6.3 Retrieving Documentation and Implementation

The detector's core logic involves retrieving the documentation and implementation of each function within a smart contract. This process primarily relies on Wake's IR model. Firstly, the detector traverses the **ir.ContractDefinition** to extract the contract's state variables and modifiers. Secondly, the detector traverses **ir.FunctionDefinition** to get the NatSpec documentation and the source code of the implemented functions. All the information is stored in the **context** variable to be then sent to LLM for further processing. Code Listings 6.2 and 6.3 show the context-building process and IR traversal.

6.4 LLM Component

The detector employs a Large Language Model to compare documentation with its implementation. This AI model produces insights and detects possible discrepancies between the two.

To improve its semantic analysis functions, the detector uses hierarchical context extraction. It assesses the contract-level context at the highest tier, concentrating on state variables and modifier declarations that establish the semantic framework. It also considers function-specific context, which pertains to the relevant applications of modifiers for the specific function under review. In addition, the detector explores the

Code listing 6.2 Building Context

```
def _process_contract_functions(self, contract: ir.ContractDefinition,
    functions: List[Tuple[ir.FunctionDefinition, Dict]]) -> None:
    if not functions:
       return
    # Build a combined context for all functions in this contract
    combined_context = f"Contract: {contract.name}\n\n"
    # Add state variables and modifiers at the beginning
    state_vars = self._extract_state_variables(contract)
    if state_vars:
       combined_context +=
         f"Contract State Variables:\n{self._compact_text(state_vars)}\n"

    all_modifiers = self._extract_all_contract_modifiers(contract)
    if all_modifiers:
       combined_context +=
         f"Contract Modifiers:\n{self._compact_text(all_modifiers)}\n"

    combined_context += "--- FUNCTIONS ---\n"
    # Now add each function
    for i, (func_node, _) in enumerate(functions):
       combined_context += f"Function #{i+1}: {func_node.name}\n"

       # Add function-specific context
       function_context = self._build_function_context(func_node)
       combined_context += function_context
       combined_context += "\n---\n"
    ...
    # Further implementation
```

Code listing 6.3 Extraction of NatSpec and Source Code

```
def _build_function_context(self, node:ir.FunctionDefinition)->str:
    doc_source = "No documentation"
    if hasattr(node, 'documentation') and node.documentation:
       if hasattr(node.documentation, 'source'):
          try:
             if isinstance(node.documentation.source, bytes):
                doc_source =
                  node.documentation.source.decode('utf-8')
             else:
                doc_source = str(node.documentation.source)
          except Exception as e:
             if hasattr(node.documentation, 'text'):
                doc_source = node.documentation.text
    impl_source = "Function implementation not available"
    if hasattr(node, 'source'):
       try:
          if isinstance(node.source, bytes):
             impl_source =
               node.source.decode('utf-8', errors='replace')
          else:
             impl_source = str(node.source)
       except Exception as e:
          impl_source =
            f"Function {node.name} ({node.visibility.name})"
    contract = self._find_parent_contract(node)
    function_modifiers_text = ""
    if contract:
       used_modifiers =
         self._extract_function_modifiers(node, contract)
       if used_modifiers:
          function_modifiers_text =
            f"Function Uses Modifiers: {used_modifiers}"
    doc_source = self._compact_text(doc_source)
    impl_source = self._compact_text(impl_source)
    context = ""
    if function_modifiers_text:
       context += f"{function_modifiers_text}\n\n"
    context +=
      f"Documentation:\n{doc_source}\nImplementation:\n{impl_source}"
    return context
```

semantic relationships among contracts and functions, acknowledging the parent-child dynamics that influence their interactions. This strategy provides an understanding of the documentation related to the implementation.

The detector optimizes performance by making a single API call per contract rather than per function to address the challenges associated with large input contexts. Feeding too much information to the language model at once can lead to loss of context and inaccurate evaluation results. By aggregating the documentation and implementation details of all functions within a contract into a single context, the detector reduces redundancy and ensures clearer contextual understanding during evaluation.

6.5 LLM Integration for Semantic Analysis

Creating the correct prompt minimizes false positives. The prompt provided to the AI model must encapsulate the essence of the function's documentation and implementation accurately.

6.5.1 Prompt Engineering Methodology

Prompt engineering techniques influence the effectiveness of applying LLMs to static analysis tasks. The detector implements a structured prompting methodology incorporating domain expertise, contextual enrichment, and false positive mitigation strategies.

In implementing prompt engineering techniques, several design patterns emerge that enhance the effectiveness of applying LLMs to static analysis tasks.

One key aspect is establishing the LLM's role, which positions it as a domain expert through explicit role attribution, such as instructing it to "ACT as an expert Solidity auditor."

Contextual framing plays a vital role by presenting structured information, facilitating a better understanding of hierarchical relationships within the analyzed content. A clear definition of the task is also important, ensuring the analytical objectives are defined without ambiguity.

Moreover, incorporating domain-specific heuristics provides guidelines that reflect patterns and edge cases relevant to Solidity analysis, enriching the context further.

Setting response formatting requirements by defining an explicit output schema allows for the parsing of the LLM's responses. These elements create a framework for effectively leveraging LLMs in static analysis tasks. [27]

6.5.2 Query Optimization Techniques

The detector implements query optimization techniques to mitigate computational resource constraints and API rate limitations.

Several optimization strategies can enhance performance. One key approach is response caching, which involves implementing an in-memory cache to avoid making redundant API calls for identical contexts. It not only saves time but also resources.

Another strategy is contract-level batching, where multiple function analyses are aggregated into a single query to the language model, reducing the volume of API calls.

Context deduplication also plays a crucial role. We streamline operations by extracting shared contextual elements, such as state variables and modifiers, once per contract instead of repeating the process for each function.

Lastly, employing exponential backoff as a form of retry logic handles any transient API failures or issues with rate limiting, as it progressively increases the delay between retries. We can achieve more efficient and reliable API interactions by integrating these strategies.

6.5.3 False Positive Reduction

A challenge in applying LLMs to static analysis is the tendency to produce false positives. The detector employs several mechanisms to mitigate this risk.

Key strategies for reducing false positives include several approaches. First, exemplar-based instruction plays a crucial role by providing explicit counter-examples that illustrate scenarios that should not be flagged as mismatches. This method helps clarify what constitutes an actual negative case.

Another important strategy is semantic equivalence recognition, which offers guidelines for identifying when syntactically different constructs convey the same semantic meaning. This recognition enables a better understanding and categorization of data that may appear dissimilar at first glance.

Response temperature control allows for the utilization of deterministic inference by adjusting the temperature parameter. This setting can mitigate ambiguities and improve response accuracy. [28]

Modifier analysis focuses on extracting and providing function modifiers, which enrich the understanding of the behavioral context involved in decision-making. By considering these modifiers, one can obtain a more comprehensive view of the data, reducing the likelihood of false positives.

This architecture demonstrates the application of domain-specific knowledge to constrain the LLM within appropriate boundaries for static analysis tasks. It enables high-confidence detection of semantic mismatches while minimizing findings that would reduce practitioner trust.

6.5.4 Performance Considerations

Integrating the LLM presents several performance trade-offs that justify careful consideration in academic analysis. First, the LLM API's token usage is noteworthy; the detector's batching strategy enhances economic efficiency by reducing total token consumption and eliminating redundant context transmission.

Moreover, there is a relationship between latency and throughput. While larger batch sizes may increase the latency of individual queries, they ultimately contribute to improved throughput by minimizing the total overhead of API calls.

The adoption of the Claude 3.7 Sonnet model reflects a balance between analytical depth and computational efficiency, establishing it as the leading choice for understanding source code. It has consistently provided reliable and accurate results. It is also important to highlight the trade-off between determinism and creativity. By setting the temperature parameter to zero, the system focuses on producing reproducible results, which may restrict the exploration of more unusual edge cases.

This methodology represents an advancement over traditional regex or AST-based documentation validation approaches. It enables semantic understanding that more closely approximates human code review practices while maintaining computational scalability.

6.6 Collecting Detections from LLM Response

The implementation of a Wake framework detector requires adherence to the detector API specification, which mandates that each detector must return a list of **DetectorResult** objects. This requirement establishes a standardized interface for all vulnerability detection implementations, enabling consistent handling and presentation of findings within the Wake ecosystem.

6.6.1 Wake Detector API Requirements

The Wake framework employs a structured approach to represent security findings through two key classes:

Code listing 6.4 Detectors API Snippet

```
@dataclass(eq=True, frozen=True)
class Detection:
    ir_node: IrAbc
    message: str
    subdetections: Tuple[Detection, ...] =
      field(default_factory=tuple)
    lsp_range: Optional[Tuple[int, int]] = field(default=None)
    subdetections_mandatory: bool = field(default=True)

@dataclass(eq=True, frozen=True)
class DetectorResult:
    detection: Detection
    impact: DetectorImpact
    confidence: DetectorConfidence
    uri: Optional[str] = field(default=None)

@total_ordering
class DetectorConfidence(StrEnum):
    LOW = "low"
    MEDIUM = "medium"
    HIGH = "high"

@total_ordering
class DetectorImpact(StrEnum):
    INFO = "info"
    WARNING = "warning"
    LOW = "low"
    MEDIUM = "medium"
    HIGH = "high"
```

The Detection class binds a security finding to a specific location in the source code through an IR node, along with a message explaining the issue. The **DetectorResult** class wraps this detection with additional metadata, including impact severity and confidence levels, which are crucial for prioritizing security findings.

6.6.2 Processing LLM Responses

The '_process_llm_response' function shown in Code Listing 6.5 serves as a link between the unstructured outputs produced by the LLM and the structured format required by the Wake framework for detection purposes. It performs several tasks to facilitate the effective communication of function analysis results.

The function normalizes and parses the LLM response by breaking it down into individual lines. For each function, it looks for the corresponding line in the LLM output using a predefined pattern, such as "Function #N:". This approach relies on prompt engineering, ensuring that the LLM delivers responses in a consistent and interpretable format.

The implementation uses a binary classification system to categorize each function analysis as either `[MATCH]` or `[MISMATCH]`. This standard labeling simplifies the parsing process and provides indicators regarding which functions have documentation inconsistencies that require attention.

When a mismatch is identified, the function extracts the explanatory portion from the response string. This explanation forms the core of the detection message, offering valuable context to the developer about the reasons behind the inconsistency between the documentation and the implementation.

6.6.3 Detection Object Generation

For each identified documentation mismatch, the function creates a Wake **Detection** object with several key components.

Each finding is closely tied to the function's IR node in the detection process, creating a direct link between the identified issue and its corresponding location in the source code. The explanation provided by the LLM serves as the detection message, offering insights into the nature of the documentation mismatch encountered.

To aid developers, the function utilizes the ‘name_location‘ attribute of the function node, indicating where the issue exists in the source code. This feature enables highlighting within IDEs, making it easier for users to identify the problem.

Furthermore, every detection is encapsulated within a **DetectorResult**. This structure includes predefined impact and confidence levels, which have been established based on the severity of the identified documentation inconsistencies. This approach ensures that users receive clear information regarding the detected issues.

The resulting list of **DetectorResult** objects is accumulated in the class' ‘_detections‘ attribute, which is later returned by the ‘detect‘ method to integrate with the Wake framework's reporting and visualization mechanisms.

6.7 Reputational Risks of Findings

Inconsistencies between documentation and implementation do not necessarily indicate critical vulnerabilities. However, they pose reputational risks. Developers and users rely on accurate documentation to understand and interact with smart contracts. Misaligned documentation can lead to misuse, loss of trust, and potential financial loss. Continuous attention to documentation consistency can enhance a project's reliability and credibility.

Code listing 6.5 Process LLM Response

```
def _process_llm_response(self, response: str,
functions: List[Tuple[ir.FunctionDefinition, Dict]]) -> None:
    lines = response.split('\n')
    analysis_results = {}
    for i, (func_node, _) in enumerate(functions):
        function_id = f"Function #{i+1}:"
        function_name =
          func_node.name if func_node.name else f"unnamed_{i}"
        match_found = False
        for line in lines:
            if function_id in line:
                match_found = True
                has_mismatch = "[MISMATCH]" in line
                if has_mismatch:
                    explanation =
                      line.split("[MISMATCH]", 1)[1].strip()
                else:
                    explanation =
                      line.split("[MATCH]", 1)[1].strip()

                raw_analysis = line.split(":", 1)[1].strip()
                analysis_results[function_name] = raw_analysis
                if has_mismatch:
                    message = explanation
                    detection = Detection(
                       ir_node=func_node,
                       message=message,
                       subdetections=(),
                       lsp_range=func_node.name_location
                         if hasattr(func_node, 'name_location')
                         else None,
                       subdetections_mandatory=False,
                    )
                    result = DetectorResult(
                       detection=detection,
                       impact=self.IMPACT,
                       confidence=self.CONFIDENCE,
                    )
                    self._detections.append(result)
                break
        if not match_found:
            analysis_results[function_name] =
              "No analysis available from LLM"
```

6.8 Testing and Evaluation

The **documentation-diff** detector was tested to evaluate its effectiveness and cost implications, following the same methodology used for existing Wake detectors.

This detector's impact level was classified as WARNING. Although it does not directly impact potential exploits, it can cause significant issues during development due to misunderstandings and misuse. This may lead to critical problems and potential financial losses.

The confidence level was set to MEDIUM due to the possibility of false positives arising from the use of LLMs, which are still not completely reliable. However, with comprehensive prompting and context-building, we achieved stable results on the previously utilized testing set. It is important to mention that some smart contracts may not have documentation provided for their functions; these are not considered for evaluation and are skipped.

The table below shows the results of running the implemented detector:

Table 6.1 Summary of Documentation Mismatches

Contract	Impact	Confidence	Count	Type
R0AR TOKEN	WARNING	MEDIUM	4	documentation-diff
RandomDEX	WARNING	MEDIUM	4	documentation-diff
MyStandard	WARNING	MEDIUM	3	documentation-diff
E280	WARNING	MEDIUM	2	documentation-doff
TWGToken	WARNING	MEDIUM	2	documentation-diff
TORN	WARNING	MEDIUM	2	documentation-diff
CATERC20	WARNING	MEDIUM	2	documentation-diff
Meme Index	WARNING	MEDIUM	2	documentation-diff
LegalXToken	WARNING	MEDIUM	1	documentation-diff
DecentralizedEURO	WARNING	MEDIUM	1	documentation-diff

6.8.1 Evaluation of Results

To determine the overall risk exposure, we obtain the liquidity values of the base tokens in question. The results are compiled in Table 6.2, reflecting the liquidity of each affected contract, following the same approach as before.

Table 6.2 Affected Value of Contracts

Contract	Type	TVL
R0AR TOKEN	documentation-diff	$1,215,257.45
RandomDEX	documentation-diff	$8,300.00
MyStandard	documentation-diff	$45,500.00
E280	documentation-diff	$12,354.78
TWGToken	documentation-diff	$25,564.30
TORN	documentation-diff	$298,700.15
CATERC20	documentation-diff	$8,034.76
Meme Index	documentation-diff	$62,796.43
LegalXToken	documentation-diff	$41,481.91
DecentralizedEURO	documentation-diff	$105,464.80

6.8.2 Detailed Examination

1. **Contract**: R0AR TOKEN
   Line Numbers: 877, 1115, 1153, 1321
   Description:

   - **Constructor Documentation Gap:** The constructor documentation fails to mention that it sets the total supply and transfers initial balances to specific addresses, which are critical token initialization behaviors. This omission could mislead developers about the token's initial distribution structure.
   - **Undocumented increaseAllowance Restriction:** The increaseAllowance function includes an onlyOwner modifier that restricts its use to the contract owner, but this critical access control is undocumented. This deviation from standard ERC20 behavior, where allowance functions are available to all users, could cause integration failures.
   - **Undocumented decreaseAllowance Restriction:** Similarly, the decreaseAllowance function is restricted to the contract owner without mentioning this limitation in the documentation. Users expecting standard ERC20 allowance functionality would be surprised to find their transactions reverting when attempting to decrease allowances.
   - **Incomplete Transfer Functionality Documentation:** The _transfer documentation only mentions tokenomics customization but omits the critical trading restrictions implemented through the tradingOpen check. This hidden functionality allows the contract to completely disable token transfers under certain conditions, impacting user interactions with the token.

   Estimated Financial Impact: $1,215,257.45

2. **Contract**: RandomDEX
   Line Numbers: 253, 64, 113, 126
   Description:

   - **TransferFrom Access Control Mismatch:** The transferFrom function documentation states it is restricted to DEFAULT_ADMIN_ROLE holders, but the implementation allows additional users with the role ALLOWED_TRANSFER_FROM_ROLE to call it before the listing timestamp. This undocumented permission creates a security gap where unexpected addresses may have transfer capabilities during the pre-listing phase.
   - **Constructor Validation Omissions:** The constructor documentation fails to document critical validation checks for zero addresses, the relationship between the maximum numerator and denominator, and the constraint that antibot fees cannot exceed the denominator. These undocumented validations could cause unexpected transaction reversions when deploying the contract with parameters that do not meet these undisclosed requirements.

- **Antibiot Timestamp Logic Error:** The updateAntibotEndTimestamp function contains a logic error where it checks if the current timestamp (antibotEndTimestamp) is less than the block.timestamp instead of validating the new value (antibotEndTimestamp_). This implementation bug contradicts the function's purpose of setting a new timestamp and may cause unexpected reverts when attempting to update this value.
- **Fee Computation Documentation Gaps:** The _computeFee function documentation omits two critical behaviors: that admin role holders are completely exempt from fees and how antibot fees are calculated based on the current timestamp. These undocumented fee exceptions and calculations make it impossible for users to accurately predict transaction costs from the documentation.

Estimated Financial Impact: $8,300.00

3. **Contract**: MyStandard
   Line Numbers: 103, 115, 1417
   Description:

   - **TransferOwnership Documentation Omission:** The transferOwnership function documentation only describes the parameter but fails to explain that this function initiates a two-step ownership transfer process rather than directly transferring control. This omission is significant because users would naturally expect an immediate ownership change when calling this function, but would be confused when that does not happen without further actions, leaving contracts in an unexpected governance state.
   - **Private TransferOwnership Implementation Gap:** The private function's _transferOwnership documentation similarly only describes its parameter without explaining its actual behavior of setting up a pending owner rather than immediately transferring control. The implementation contains conditional logic that handles different ownership transfer scenarios based on whether the recipient is the current sender. Still, none of this critical ownership management flow is documented, making maintenance and security reviews more difficult.
   - **BurnFrom Documentation Typo:** The burnFrom function documentation contains a minor grammatical inconsistency where it refers to "accounts" (plural) instead of "account" (singular) in the requirements section. At the same time, the implementation correctly uses the singular form. While this is a minor issue that does not affect functionality, it represents an inconsistency in the documentation quality that could confuse users, especially when compared with other similar function descriptions.

   Estimated Financial Impact: $45,500.00

4. **Contract**: E280
   Line Numbers: 75, 90
   Description:

- **MintWithElmnt Function Documentation Gap:** The mintWithElmnt function documentation mentions transforming "user's ELMNT into E280" but fails to specify that ELMNT refers to a specific token contract address that will be called during execution. It also omits the critical precondition that the mintingEnabled flag must be true for this function to work, potentially leading to unexpected transaction failures when this global state variable is disabled.
- **Distribute Function Vague Documentation:** The distribute function has a highly brief documentation stating it "Distributes ELMNT from mints to its destinations" without explaining the complex allocation logic implemented in the code. The function performs multiple distribution operations to different addresses with specific ratios. It includes an incentive fee mechanism, which is not documented, making it impossible for users or auditors to understand the economic implications of calling this function.

Estimated Financial Impact: $12,354.78

5. **Contract**: TWGToken
   Line Numbers: 73, 238
   Description:

   - **Transfer Function Insufficient Documentation:** The _transfer function documentation is minimal, stating that it "checks restrictions and apply tax" while omitting critical operational details. The implementation contains complex logic for transaction limits, trading status validations, and tax calculations that affect transaction behavior but are undocumented. This insufficient documentation makes it difficult for developers to understand when transfers might fail or have fees applied, leading to integration issues or unexpected token behavior.
   - **AddLiquidity Missing Parameter Documentation:** The addLiquidity function documentation completely omits the required parameter _uniswapRouter, despite this parameter being necessary for the implementation. The documentation only describes the tokenAmount parameter while ignoring this critical address parameter, which determines which Uniswap router contract will receive the liquidity. This omission could lead to incorrect function calls or confusion when interacting with the contract, especially since the router choice impacts where the token's liquidity will be established.

   Estimated Financial Impact: $25,564.30

6. **Contract**: TORN
   Line Numbers: 1535, 2761
   Description:

   - **BurnFrom Documentation Typo:** The burnFrom function documentation contains a minor inconsistency in that it refers to "accounts" (plural) in the requirements section while the implementation parameter is correctly named "account" (singular). This is a relatively minor documentation issue that does not affect functionality but could cause confusion when reading the code, particularly for developers who rely on precise documentation. Despite this grammatical error, the actual implementation behavior correctly matches the documented functionality.

- **RescueTokens Insufficient Documentation:** The rescueTokens function has severely inadequate documentation, describing it only as a "Method to claim junk and accidentally sent tokens" while omitting numerous critical aspects of its behavior. The implementation includes governance-only access restrictions, support for both token and ETH recovery, specific balance parameter handling, zero address validation checks, and non-zero balance requirements, none of which are mentioned in the documentation. These omissions hinder the ability of developers or auditors to understand the function's behavior, security model, and usage constraints without reading the implementation code in detail.

Estimated Financial Impact: $298,700.15

7. **Contract**: CATERC20
Line Numbers: 15, 23
Description:

- **SplitSignature Documentation Inadequacy:** The splitSignature function has critically insufficient documentation, consisting only of the vague phrase "signature methods" without explaining its purpose, expected input format, or output values. The function performs the essential cryptographic operation of decomposing an Ethereum signature into its standard components (v, r, s). It requires a specific 65-byte input format as enforced by the code's validation check. This documentation gap makes it difficult for developers to properly use this security-critical function without studying the implementation details.
- **BridgeOut Documentation Omissions:** The bridgeOut function documentation is severely lacking, stating only "To bridge tokens to other chains" without explaining any of its parameters, payable nature, or return values. The function implements cross-chain token bridging functionality with multiple critical parameters (amount, recipientChain, recipient, nonce) that affect where and how tokens are transferred, yet does not guide proper parameter usage, security considerations, or the meaning of the returned sequence value. These omissions make the contract's cross-chain functionality difficult to use correctly and safely.

Estimated Financial Impact: $8,034.76

8. **Contract**: Meme Index
Line Numbers: 473, 797
Description:

- **Constructor Documentation Inadequacy:** The constructor documentation states "Contract constructor" without mentioning any of the critical token initialization steps performed in the implementation. The code sets the token name to

"Meme Index", configures the token symbol, and distributes the initial token supply by minting to five different addresses, all constituting fundamental aspects of the token's economic design that should be appropriately documented for transparency and auditability.

- **Transfer Zero Amount Restriction:** The _transfer function documentation fails to mention a vital input validation requirement that the amount parameter must be greater than zero, which is strictly enforced in the implementation. This undocumented restriction could cause unexpected transaction failures for integrations that attempt to make zero-value transfers (which are valid in some ERC20 implementations), breaking dependent contracts or user interfaces that have not accounted for this non-standard behavior.

Estimated Financial Impact: $62,796.43

9. **Contract**: LegalXToken
Line Numbers: 2497
Description: The withdrawDividend() function documentation contains a technical inaccuracy regarding event emission behavior. The documentation states "It emits a DividendWithdrawn event if the amount of withdrawn ether is greater than 0," implying that this function directly emits the event. However, the implementation shows that withdrawDividend() does not emit any events directly - it simply delegates to the internal function _withdrawDividendOfUser(), which is presumably where the event emission occurs. While the result is the same (the event gets emitted during the function call's execution flow), the documentation is misleading about which function in the call stack is responsible for the event emission, confusing for developers auditing or maintaining the code.

   Estimated Financial Impact: $41,481.91

10. **Contract**: DecentralizedEURO
Line Numbers: 147
Description: The _adjustTotalVotes function documentation has a critical parameter omission. While the implementation requires three parameters (from, amount, and roundingLoss) and uses all three in its vote calculation logic, the documentation only documents two parameters, entirely omitting the roundingLoss parameter. This parameter is not just supplementary but plays a key role in the voting mechanism, as it is directly subtracted from the total votes calculation (totalVotes() - roundingLoss - lostVotes). This documentation gap could lead to incorrect usage of the function by developers extending this contract, potentially causing unexpected voting power adjustments since they would not be aware of the need to accurately calculate and provide the roundingLoss value when calling this internal function.

    Estimated Financial Impact: $105,464.80

6.9 Summary

The cost of potential exploits due to documentation inconsistencies can be significant. Reputational damage can lead to loss of user trust and financial repercussions. Accurate documentation minimizes these risks, ensuring that smart contracts operate as intended.

- Inaccurate documentation can mislead users and developers.
- Consistent documentation enhances transparency and trust.
- Regular audits and updates to documentation can mitigate reputational risks.

By applying the implemented methodology and the **documentation-diff** detector, the financial impact of possible reputational damage sums up to **$1,823,454.58**.

The `documentation-diff` detector represents an approach to automated documentation validation in smart contracts, combining static analysis techniques with natural language processing. This integration enables the detection of semantic inconsistencies through syntactic analysis, enhancing both development efficiency and contract security.

This approach could be extended to other static analysis domains requiring semantic understanding beyond syntactic correctness.

Conclusion

This thesis aimed to assess the financial impact of Ethereum vulnerability detectors by analyzing their effectiveness and potential cost savings. The research encompassed several phases, from collecting Ethereum smart contract data to developing and testing a new detector within the Wake framework.

Data Collection and Enrichment

Part of the thesis involved collecting smart contract data from the Ethereum mainnet. The methodology ensured the dataset was relevant and representative of real-world conditions, highlighting contracts with noteworthy assets and active engagement.

Evaluation of Existing Detectors

The evaluation section focused on existing vulnerability detectors' ability to prevent economic exploits. The thesis quantified the detectors' financial impact by estimating potential savings from preventative measures. It involved a detailed analysis of the cost of exploits relative to the assets held by affected contracts, demonstrating the benefits of timely alerts.

Detector Development

A new detector was developed and integrated into the Wake framework. This detector aims to identify inconsistencies between smart contract documentation and implementation. By leveraging AI for evaluation, the detector may enhance code quality and mitigate mismatches. Testing against real-world contracts validates its functionality and effectiveness.

Testing and Economic Assessment

The effectiveness of the developed detector was tested using a selection of contracts. An economic assessment evaluates the financial value protected by identifying vulnerabilities. This assessment highlighted the financial advantages of incorporating advanced security measures into the contract development process.

Practical Relevance and Application

The thesis demonstrates the importance of defining a testing set of contracts for running vulnerability detectors. A toolkit was developed to parse and filter recently deployed contracts, ensuring a focus on those with significant value and active participation. This method provided a practical approach to evaluating security measures' real-world applicability and impact.

Implications and Future Work

The critical role of vulnerability detectors in securing Ethereum smart contracts was highlighted. These detectors contribute to a secure decentralized ecosystem by reducing financial losses and enhancing trust in blockchain technology. The documentation-diff detector developed in this thesis represents a step forward, but further advancements are necessary. Future work could explore optimizing the detector's algorithms, expanding its applicability to other blockchain platforms, and refining the economic assessment models for a more precise estimation of cost savings.

Concluding Remarks

In conclusion, this thesis underlines the importance of integrating vulnerability detectors in the Ethereum ecosystem. The findings demonstrate that such detectors safeguard against potential exploits and provide financial benefits. Therefore, the continuous evolution and implementation of security measures are imperative for blockchain technologies' sustainable growth and trustworthiness.

Appendix A
Code Listings

Code listing A.1 Full _check_reentrancy_in_function

```
def _check_reentrancy_in_function(
        self,
        function_definition: ir.FunctionDefinition,
        statement: ir.StatementAbc,
        address_source: ir.ExpressionAbc,
        child_modifies_state: Set[Tuple[ir.IrAbc, analysis.
           ModifiesStateFlag]],
        checked_statements: Set[ir.StatementAbc],
    ) -> List[Tuple[Detection, DetectorImpact,
       DetectorConfidence]]:
        from functools import reduce
        from operator import or_

        from wake.analysis.expressions import
           find_low_level_call_source_address
        from wake.analysis.ownable import (
            address_is_safe,
            statement_is_publicly_executable,
        )
        from wake.analysis.utils import (
            get_all_base_and_child_declarations,
            pair_function_call_arguments,
        )

        # TODO check non-reentrant
        if not statement_is_publicly_executable(statement,
           check_only_eoa=True):
            return []
```

```
source_address_declaration =
   find_low_level_call_source_address(address_source)
is_safe = None
if source_address_declaration is None:
    pass
    # self.logger.debug(f"{address_source.source}")
elif isinstance(source_address_declaration, ir.enums.
   GlobalSymbol):
    if source_address_declaration == ir.enums.
       GlobalSymbol.THIS:
        is_safe = True
    elif source_address_declaration in {
        ir.enums.GlobalSymbol.MSG_SENDER,
        ir.enums.GlobalSymbol.TX_ORIGIN,
    }:
        is_safe = False
    else:
        is_safe = None
        # self.logger.debug(f"{
           source_address_declaration}:")
elif isinstance(source_address_declaration, ir.
   ContractDefinition):
    if source_address_declaration.kind == ir.enums.
       ContractKind.LIBRARY:
        is_safe = True
elif isinstance(source_address_declaration, ir.Literal
   ):
    is_safe = True
else:
    is_safe = address_is_safe(
       source_address_declaration)

if is_safe:
    return []

checked_statements.add(statement)
ret = []

this_modifies_state = set(child_modifies_state)
this_modifies_state.update(
    _modifies_state_after_statement(
       function_definition, statement)
)

if len(this_modifies_state) and function_definition.
   visibility in {
    ir.enums.Visibility.PUBLIC,
    ir.enums.Visibility.EXTERNAL,
}:
    state_mods = reduce(or_, (mod[1] for mod in
       this_modifies_state))
    if state_mods & (
        analysis.ModifiesStateFlag.MODIFIES_STATE_VAR
        | analysis.ModifiesStateFlag.SENDS_ETHER
        | analysis.ModifiesStateFlag.PERFORMS_CALL
        | analysis.ModifiesStateFlag.
           CALLS_UNIMPLEMENTED_NONPAYABLE_FUNCTION
        | analysis.ModifiesStateFlag.
           CALLS_UNIMPLEMENTED_PAYABLE_FUNCTION
    ):
        impact = DetectorImpact.HIGH
    elif state_mods & (
        analysis.ModifiesStateFlag.EMITS
        | analysis.ModifiesStateFlag.DEPLOYS_CONTRACT
        | analysis.ModifiesStateFlag.SELFDESTRUCTS
        | analysis.ModifiesStateFlag.
           PERFORMS_DELEGATECALL
    ):
        impact = DetectorImpact.WARNING
    else:
        raise NotImplementedError()

    ret.append(
        (
            Detection(
                statement,
                f"Exploitable from `{
                   function_definition.canonical_name
                   }`",
            ),
            impact,
            DetectorConfidence.LOW
            if is_safe is None
            else DetectorConfidence.MEDIUM,
        )
    )

for ref in function_definition.get_all_references(
   False):
    if isinstance(ref, ir.IdentifierPathPart):
        top_statement = ref.underlying_node
    elif isinstance(ref, ir.ExternalReference):
        continue  # TODO currently not supported
    else:
        top_statement = ref
    func_call = None
    while top_statement is not None:
        if (
            func_call is None
            and isinstance(top_statement, ir.
               FunctionCall)
            and top_statement.function_called
            in get_all_base_and_child_declarations(
               function_definition)
        ):
            func_call = top_statement
        if isinstance(top_statement, ir.StatementAbc):
            break
        top_statement = top_statement.parent

    if top_statement is None or func_call is None:
        continue
    function_def = top_statement
    while function_def is not None and not isinstance(
        function_def, ir.FunctionDefinition
    ):
        function_def = function_def.parent
    if function_def is None:
        continue
    assert isinstance(function_def, ir.
       FunctionDefinition)
    if top_statement in checked_statements:
        continue

    if source_address_declaration in
       function_definition.parameters.parameters:
        for arg_decl, arg_expr in
           pair_function_call_arguments(
            function_definition, func_call
        ):
            if arg_decl == source_address_declaration:
                assert isinstance(
                    arg_expr.type, (types.Address,
                       types.Contract)
                )
                ret.extend(
                    self._check_reentrancy_in_function
                       (
                                function_def,
                                top_statement,
                                arg_expr,
                                this_modifies_state,
                                checked_statements,
                            )
                        )
                        break
                else:
                    ret.extend(
                        self._check_reentrancy_in_function(
                            function_def,
                            top_statement,
                            address_source,
                            this_modifies_state,
                            checked_statements,
                        )
                    )
        return ret
```

Code listing A.2 Full TxOriginDetector Class

```
class TxOriginDetector(Detector):
    _account_abstraction: bool
    _detections: List[DetectorResult]

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

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

    def visit_member_access(self, node: ir.MemberAccess):
        from wake.analysis.expressions import
           expression_is_global_symbol

        if node.referenced_declaration != ir.enums.
           GlobalSymbol.TX_ORIGIN:
            return

        if self._account_abstraction:
            self._detections.append(
                DetectorResult(
                    Detection(
                        node,
                        "Use of tx.origin may interfere with
                           ERC-4337 account abstraction",
            ),
            impact=DetectorImpact.WARNING,
            confidence=DetectorConfidence.LOW,
            uri=generate_detector_uri(
                name="tx-origin",
                version=self.extra["package_versions"
                   ]["eth-wake"],
                anchor="account-abstraction",
            ),
        )
    )

np = node.parent
npp = node.parent.parent
if np is not None and npp is not None:
    if (
        isinstance(np, ir.BinaryOperation)
        and np.operator == ir.enums.BinaryOpOperator.
           EQ
    ):
        other_expr = (
            np.right_expression
            if np.left_expression == node
            else np.left_expression
        )
        if expression_is_global_symbol(
            other_expr, ir.enums.GlobalSymbol.
               MSG_SENDER
        ):
            return

    elif isinstance(np, ir.IndexAccess):
        if isinstance(npp, ir.BinaryOperation) and npp
           .operator in {
            ir.enums.BinaryOpOperator.LT,
            ir.enums.BinaryOpOperator.GT,
        }:
            other_expr = (
                npp.right_expression
                if npp.left_expression == np
                else npp.left_expression
            )
            if expression_is_global_symbol(
                other_expr, ir.enums.GlobalSymbol.
                   BLOCK_TIMESTAMP
            ):
```

```
                        return

        self._detections.append(
            DetectorResult(
                Detection(node, "Unsafe usage of tx.origin"),
                impact=DetectorImpact.MEDIUM,
                confidence=DetectorConfidence.LOW,
                uri=generate_detector_uri(
                    name="tx-origin",
                    version=self.extra["package_versions"]["
                       eth-wake"],
                    anchor="phishing-attacks",
                ),
            )
        )

    @detector.command(name="tx-origin")
    @click.option(
        "--account-abstraction/--no-account-abstraction",
        is_flag=True,
        default=True,
        help="Report account abstraction related issues.",
    )
    def cli(self, account_abstraction: bool) -> None:
        """
        Possibly incorrect usage of tx.origin
        """
        self._account_abstraction = account_abstraction
```

Code listing A.3 Full UnsafeDelegatecallDetector Class

```
class UnsafeDelegatecallDetector(Detector):
    _proxy: bool
    _detections: List[DetectorResult]

    def __init__(self):
        self._detections = []

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

    def visit_member_access(self, node: ir.MemberAccess):
        from wake.analysis.proxy import contract_is_proxy

        t = node.type
        if (
            not isinstance(t, types.Function)
        or t.kind
        not in {
            ir.enums.FunctionTypeKind.DELEGATE_CALL,
            ir.enums.FunctionTypeKind.BARE_DELEGATE_CALL,
        }
        or t.attached_to is not None
        or node.statement is None
    ):
        return

    func = node.statement.declaration
    # TODO: delegatecalls in modifiers
    if not isinstance(func, ir.FunctionDefinition):
        return

    contract = func.parent
    if (
        not self._proxy
        and isinstance(contract, ir.ContractDefinition)
        and contract_is_proxy(contract)
    ):
        return

    ret = check_delegatecall_in_function(
        func, node.statement, node.expression, set()
    )
    if len(ret) == 0:
        return

    self._detections.append(
        DetectorResult(
            Detection(
                node,
                f"Possibly unsafe delegatecall in `{func.
                   canonical_name}`",
                tuple(r[0] for r in ret),
            ),
            confidence=max(r[1] for r in ret),
            impact=DetectorImpact.MEDIUM,
            uri=generate_detector_uri(
                name="unsafe-delegatecall",
                version=self.extra["package_versions"]["
                   eth-wake"],
            ),
        )
    )
    @detector.command(name="unsafe-delegatecall")
    @click.option(
        "--proxy/--no-proxy",
        is_flag=True,
        default=False,
        help="Detect delegatecalls in proxy contracts.",
    )
    def cli(self, proxy: bool) -> None:
        """
        delegatecall to untrusted contract
        """
        self._proxy = proxy
```

■ **Code listing A.4** Full _check_delegatecall_in_function

```
def check_delegatecall_in_function(
    function_definition: ir.FunctionDefinition,
    statement: ir.StatementAbc,
    address_source: ir.ExpressionAbc,
    checked_statements: Set[ir.StatementAbc],
) -> List[Tuple[Detection, DetectorConfidence]]:
    from wake.analysis.expressions import
       find_low_level_call_source_address
    from wake.analysis.ownable import address_is_safe,
       statement_is_publicly_executable
    from wake.analysis.utils import
       pair_function_call_arguments

    if not statement_is_publicly_executable(statement):
        return []

    source_address_declaration =
       find_low_level_call_source_address(address_source)
    is_safe = None
    if source_address_declaration is None:
        pass
        # logger.debug(f"{address_source.source}")
    elif isinstance(source_address_declaration, ir.enums.
       GlobalSymbol):
        if source_address_declaration == ir.enums.GlobalSymbol
            .THIS:
            is_safe = True
        elif source_address_declaration in {
            ir.enums.GlobalSymbol.MSG_SENDER,
            ir.enums.GlobalSymbol.TX_ORIGIN,
        }:
            is_safe = False
        else:
            is_safe = None
            # logger.debug(f"{source_address_declaration}:")
    elif isinstance(source_address_declaration, ir.
       ContractDefinition):
        if source_address_declaration.kind == ir.enums.
           ContractKind.LIBRARY:
            is_safe = True
    elif isinstance(source_address_declaration, ir.Literal):
        is_safe = True
    else:
        is_safe = address_is_safe(source_address_declaration)

    if is_safe:
        return []

    checked_statements.add(statement)
    ret = []
    if function_definition.visibility in {
        ir.enums.Visibility.PUBLIC,
        ir.enums.Visibility.EXTERNAL,
    }:
        ret.append(
            (
                Detection(
                    statement,
                    f"Exploitable from `{function_definition.
                       canonical_name}`",
                ),
                DetectorConfidence.LOW
                if is_safe is None
                else DetectorConfidence.MEDIUM,
            )
        )

    for ref in function_definition.get_all_references(False):
        if isinstance(ref, ir.IdentifierPathPart):
            top_statement = ref.underlying_node
        elif isinstance(ref, ir.ExternalReference):
            continue  # TODO currently not supported
        else:
            top_statement = ref
        func_call = None
        while top_statement is not None:
            if (
                func_call is None
                and isinstance(top_statement, ir.FunctionCall)
                and top_statement.function_called ==
                    function_definition
            ):
                func_call = top_statement
            if isinstance(top_statement, ir.StatementAbc):
                break
            top_statement = top_statement.parent
        if top_statement is None or func_call is None:
            continue
        function_def = top_statement
        while function_def is not None and not isinstance(
            function_def, ir.FunctionDefinition
        ):
            function_def = function_def.parent
        if function_def is None:
            continue
        assert isinstance(function_def, ir.FunctionDefinition)
        if top_statement in checked_statements:
            continue
        if source_address_declaration in function_definition.
           parameters.parameters:
            for arg_decl, arg_expr in
               pair_function_call_arguments(
                function_definition, func_call
            ):
                if arg_decl == source_address_declaration:
                    assert isinstance(arg_expr.type, (types.
                       Address, types.Contract))
                    ret.extend(
                        check_delegatecall_in_function(
                            function_def, top_statement,
                               arg_expr, checked_statements
                        )
                    )
                    break
        else:
            ret.extend(
                check_delegatecall_in_function(
                    function_def, top_statement,
                       address_source, checked_statements
                )
            )

    return ret
```

Bibliography

1. ACKEE BLOCKCHAIN. *Wake Framework*. 2024. Available also from: `https://ackee.xyz/wake/docs/latest/`. Online documentation.
2. DANNEN, Chris; DANNEN, Chris. Solidity programming. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. 2017.
3. BÖHME, Rainer; CHRISTIN, Nicolas; EDELMAN, Benjamin; MOORE, Tyler. Bitcoin: Economics, technology, and governance. *Journal of economic Perspectives*. 2015.
4. MCKINSEY & COMPANY. *What is blockchain?* 2024. Available also from: `https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-blockchain`. Article.
5. IQBAL, Mubashar; MATULEVIČIUS, Raimundas. Exploring Sybil and Double-Spending Risks in Blockchain Systems. *IEEE Access*. 2021, vol. PP. Available from DOI: `10.1109/ACCESS.2021.3081998`.
6. GHIMIRE, Suman; SELVARAJ, Henry. A Survey on Bitcoin Cryptocurrency and its Mining. In: 2018. Available from DOI: `10.1109/ICSENG.2018.8638208`.
7. LASHKARI, Bahareh; MUSILEK, Petr. A comprehensive review of blockchain consensus mechanisms. *IEEE access*. 2021.
8. GHOSH, Ananya; SARKAR, Indranil; DEY, Mrittika; GHOSH, Ahona. Artificial intelligence and blockchain: Implementation perspectives for healthcare beyond 5G. In: 2022. Available from DOI: `10.1016/B978-0-323-90615-9.00003-7`.
9. NAKAMOTO, Satoshi. *Bitcoin: A Peer-to-Peer Electronic Cash System* [White paper]. 2008. Available also from: `https://bitcoin.org/bitcoin.pdf`.
10. LIU, Debin; CAMP, L Jean. Proof of Work can Work. In: *WEIS*. Citeseer, 2006.
11. ETHEREUM DEVELOPERS. *Solidity*. 2025. Available also from: `https://docs.soliditylang.org/en/v0.8.29/`. Online documentation.
12. GRAMLICH, Benjamin. Smart contract languages: A thorough comparison. *ResearchGate Preprint*. 2020.
13. SOLIDITY TEAM. *Contracts* [`https://docs.soliditylang.org/en/v0.8.29/contracts.html`]. 2025. Accessed on 2025-04-06.

14. ETHEREUM FOUNDATION. *NatSpec Format*. 2025. Available also from: `https://docs.soliditylang.org/en/latest/natspec-format.html`. Online documentation.

15. ZETZSCHE, Dirk A; ARNER, Douglas W; BUCKLEY, Ross P. Decentralized finance (defi). *Journal of Financial Regulation*. 2020.

16. BUTERIN, Vitalik et al. A next-generation smart contract and decentralized application platform. *white paper*. 2014.

17. ETHEREUM FOUNDATION. *Ethereum's Energy Consumption*. 2023. Available also from: `https://ethereum.org/en/energy-consumption/`. Article.

18. TSANKOV, Petar; DAN, Andrei; DRACHSLER-COHEN, Dana; GERVAIS, Arthur; BUENZLI, Florian; VECHEV, Martin. Securify: Practical security analysis of smart contracts. In: *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2018.

19. 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.

20. ETHEREUM FOUNDATION. *EVM architecture*. 2024. Available also from: `https://ethereum.org/content/developers/docs/gas/gas.png`. Online documentation.

21. ETHEREUM DEVELOPERS. *Intro to Ethereum*. 2025. Available also from: `https://ethereum.org/en/developers/docs/intro-to-ethereum/`. Online documentation.

22. ACKEE BLOCKCHAIN. *Working with IR - Wake Documentation*. 2024. Available also from: `https://ackee.xyz/wake/docs/latest/static-analysis/working-with-ir/`. Online documentation.

23. ACKEE BLOCKCHAIN. *Wake CFG example*. 2024. Available also from: `https://ackee.xyz/wake/docs/latest/static-analysis/printers/control-flow-graph/`. Online documentation.

24. KHIMCHENKO, Dmytro. *Ethereum vulnerability detectors*. 2024. Available also from: `https://dspace.cvut.cz/handle/10467/115684`. Bachelor thesis. Faculty of Information Technology, Czech Technical University in Prague.

25. DEXSCREENER. *DexScreener docs*. [N.d.]. Available also from: `https://docs.dexscreener.com/api/reference`. Online documentation.

26. ACKEE BLOCKCHAIN. *Wake's built-in detectors*. 2024. Available also from: `https://ackee.xyz/wake/docs/latest/static-analysis/using-detectors/`. Online documentation.

27. BOONSTRA, Lee. *Prompt Engineering*. Google/Kaggle, 2024. Available also from: `https://www.kaggle.com/whitepaper-prompt-engineering`.

28. WANG, Kelsey. *A Comprehensive Guide to LLM Temperature*. 2025. Available also from: `https://medium.com/@kelseyywang/a-comprehensive-guide-to-llm-temperature-%EF%B8%8F-363a40bbc91f`. Article.

Content of the attachment

```
 bortnikov-master-thesis.pdf ...................... Bortnikov Master Thesis
 scripts ................................................... Automation scripts
  ├─ contract_download.py
  ├─ calculate_tvl.py
  ├─ scraper.py
  ├─ wakerun.py
  └─ requirements.txt
 contracts  ..................................... Analyzed contract addresses
  ├─ eth_contracts.txt
  └─ eth_contracts_tvl.txt
 detector.zip ................................... documentation-diff detector
  ├─ __init__.py
  └─ documentation_diff.py
 README.md ........................................ Description of Directory
```