POLITECNICO DI TORINO

MASTER OF SCIENCE IN ENGINEERING AND MANAGEMENT

MASTER'S THESIS TITLE

Ethereum Smart Contract Analysis through Chat-GPT and other NLP Techniques

Academic Advisor:

Prof. Valentina GATTESCHI

Candidate:

Muhammad Usama Akhtar
(S2898413)

MARCH-2023

Abstract

The objective of this thesis was to explore the capabilities of the OpenAI ChatGPT API in analysing Ethereum smart contracts. The research involved using the API to read the text data from the smart contracts and perform both topic analysis and sentiment analysis. The results of this study showed that the ChatGPT API is capable of accurately extracting relevant information from the smart contracts and performing a meaningful analysis. The topic analysis identified the main themes discussed in the contracts, while the sentiment analysis provided insights into the emotions and opinions expressed in the text. The findings of this research have potential implications for the field of blockchain and smart contract analysis, as it demonstrates the feasibility of using advanced NLP techniques for the automated analysis of blockchain-based contracts.

The technique used in this thesis makes it easier for individuals who have no prior knowledge of smart contract vulnerabilities to understand the risks associated with them. This is achieved by utilizing the ChatGPT API to perform topic analysis and sentiment analysis on the text data extracted from the smart contracts. The topic analysis provides a high-level understanding of the key themes discussed in the contracts, while the sentiment analysis gives insight into the emotions and opinions expressed in the text. This information can be used to identify potential risks and vulnerabilities in the contracts, such as security or privacy issues, and to make informed decisions regarding their use.

Furthermore, the automation of the analysis process using the ChatGPT API significantly reduces the time and effort required to manually review and analyse the contracts. This is particularly beneficial for individuals who may not have the technical knowledge or expertise to manually review the contracts for potential vulnerabilities. By providing an easy-to-understand representation of the risks associated with smart contracts, this technique has the potential to increase transparency and trust in the use of smart contracts in various industries.

The results of this thesis demonstrate the feasibility and effectiveness of using advanced NLP techniques, such as the ChatGPT API, to analyze and understand the risks associated with Ethereum smart contracts. This approach can help to increase the accessibility and understanding of smart contracts and promote their responsible use in various industries.

Acknowledgement

I am immensely grateful to have had the opportunity to undertake this thesis under the guidance of Prof. Valentina Gatteschi, whose valuable insights and feedback have been instrumental in shaping the outcome of my work. I also want to extend my heartfelt thanks to my family and friends for their unwavering support and encouragement throughout my academic journey.

Completing this master's program in Engineering & Management from Politecnico Di Torino has been a challenging yet fulfilling experience, and I will always cherish the memories of the knowledge gained, skills honed, and friendships made along the way. I am also grateful to the faculty and staff of the university for their contribution to my academic and personal growth.

Table of Contents

List of Tables

List of Figures

Chapter 1

1. Introduction

1.1. Research Goals

This research project aims to evaluate the accuracy and effectiveness of the ChatGPT-3.5 (Davinci-003) in performing topic analysis and sentiment analysis on Ethereum smart contracts. The study will investigate potential vulnerabilities and risks associated with Ethereum smart contracts using the ChatGPT API, and compare the results with traditional manual analysis methods. A tool or framework will be developed that integrates the ChatGPT API with existing smart contract analysis tools, and its effectiveness will be evaluated in identifying and mitigating potential risks.

In addition, this project aims to explore the potential use cases of using advanced NLP techniques, such as the ChatGPT API and other NLP methods, for the automated analysis of smart contracts in other blockchain platforms and industries. Ethical and legal implications of using automated NLP techniques for the analysis of smart contracts will also be investigated, and potential biases or limitations of the approach will be identified.

The scalability and performance limitations of the ChatGPT API in analyzing large volumes of smart contract data will also be investigated, and potential solutions or optimizations will be identified. By conducting this research, we hope to provide a better understanding of the capabilities and limitations of the ChatGPT API for the analysis of smart contracts, and to identify opportunities for improving the automated analysis of smart contracts using advanced NLP techniques such as Topic Modeling and Sentiment Anlaysis.

1.2. The Problem

Understanding Ethereum smart contracts can be challenging for a layperson without experience in programming and blockchain technology. Ethereum smart contracts are written in a programming language called Solidity, which is different from natural language, making it difficult to read and understand for non-technical people.

The complexity of smart contracts can lead to vulnerabilities, such as security loopholes or logical errors. These vulnerabilities can cause smart contracts to behave unexpectedly, leading to loss of funds or other undesirable outcomes.

There are various tools available for reading and analysing smart contracts, but they also require some technical knowledge to use effectively. For example, static analysers can identify potential security issues in the code, but they require an understanding of the potential vulnerabilities that could exist. Similarly, debuggers can help identify logical errors or other issues, but they require an understanding of how the code is structured.

Another challenge for laypeople is the lack of standardization in the way smart contracts are written and documented. The naming conventions and organization of the code can vary widely from contract to contract, making it difficult to quickly understand and identify potential issues.

1.3. Overview of the Thesis Structure

The following Chapter 2 gives a literature review of the Ethereum Blockchain technology and Natural Language Processing techniques further used in this thesis. Moreover, it discusses Ethereum smart contract analysis techniques and tools. It also identifies types of vulnerabilities found in the smart contracts and how to overcome those issues.

Furthermore, Chapter 3 gives an overview of other related research papers which involve Artificial Intelligence techniques with smart contracts.

Chapter 4 discusses the core work of the thesis. It explains the proposed system realized for the research purpose of this thesis; The dataset of smart contracts is analysed through ChatGPT API and further results are used to perform Topic Modeling and Sentiment Analysis. The subsequent Chapter 5 evaluates the results and explains the outcomes of the research.

Finally, Chapter 6 summarises the whole research in a conclusion statement and talks about the future works that maybe required for further advancements in the technology.

Chapter 2

2. Preliminaries

2.1. Blockchain and its Applications

Blockchain is a distributed ledger technology that enables secure and transparent record-keeping of transactions and data. A blockchain is composed of blocks that store data, and each block is linked to the previous one through cryptographic hashing. This creates an immutable chain of records that is resistant to tampering and manipulation.

The most well-known application of blockchain technology is cryptocurrency. Bitcoin, the first decentralized cryptocurrency, was created in 2009 and operates on a blockchain [1]. In this use case, the blockchain serves as a decentralized ledger of all Bitcoin transactions. This allows for secure and transparent transfers of ownership of the cryptocurrency without the need for a central authority, such as a bank. Figure 1 explains how blocks are made once the user makes transaction on the Blockchain.

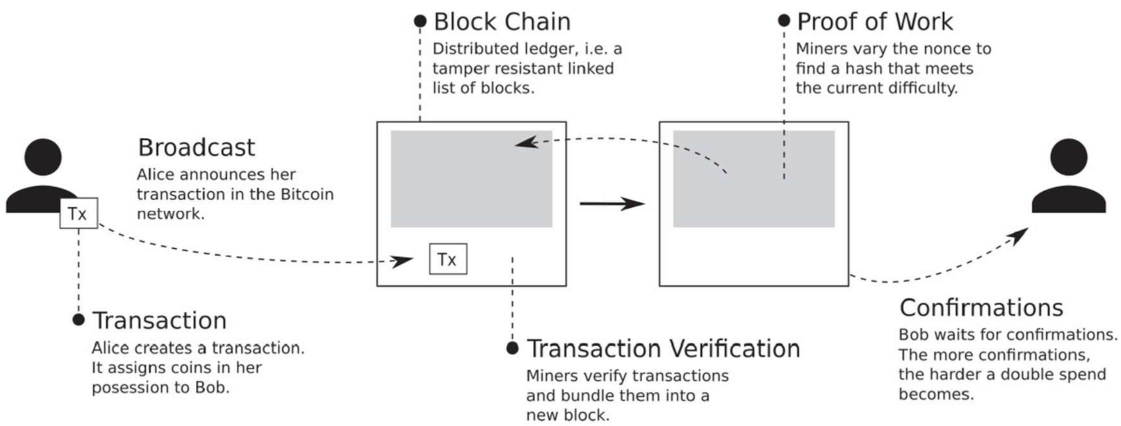


Figure 1 Block formation on bitcoin [2]

One of the important applications of blockchain technology is in supply chain management [3]. By using a blockchain to record every step of a product's journey from production to delivery, all participants in the supply chain can have a transparent view of the product's history and provenance. This helps to increase accountability, reduce fraud and improve the efficiency of the supply chain.

Blockchain technology can also be used in the voting process. By using a blockchain to record and store voting data, it becomes possible to ensure the transparency, security and accuracy of the voting process. This can help to reduce the potential for fraud and increase voter confidence in the election results [4].

Another application of blockchain technology is in the creation of decentralized autonomous organizations (DAOs) [5]. A DAO is a decentralized organization that is run through rules encoded as computer programs on a blockchain. These programs are transparent, public and cannot be altered, ensuring that the organization remains autonomous and decentralized. DAOs have the potential to disrupt traditional organizational structures by enabling decentralized decision making and increasing transparency.

In the field of digital identity, blockchain technology offers a secure and decentralized solution for storing and verifying personal information. By using a blockchain to store personal data, individuals have more control over their personal information and can ensure that it is being used in a secure and transparent manner. This can help to reduce instances of identity theft and fraud [6].

Finally, blockchain technology can be used in the creation of digital assets, such as collectibles and non-fungible tokens (NFTs). By using a blockchain to record the ownership and provenance of these digital assets, it becomes possible to ensure the authenticity and scarcity of the asset. This has led to the creation of new markets for digital collectibles and NFTs, as well as new opportunities for artists, musicians, and other creators to monetize their digital creations [7].

2.2. Smart Contracts

Smart contracts are self-executing contracts with the terms of the agreement between buyer and seller being directly written into lines of code. They are stored on the blockchain network, which is a decentralized and distributed ledger technology, making them transparent, secure and tamper-proof [8].

Smart contracts allow for the automation of various processes and the enforcement of specific terms and conditions without the need for intermediaries. This eliminates the possibility of fraud or interference and ensures that the terms of the contract are executed exactly as intended.

The code and the agreements contained in a smart contract are executed automatically when certain predetermined conditions are met. For example, in a real estate transaction, the ownership of the property could be transferred automatically to the buyer when the agreed-upon payment is received. The terms of the contract are defined in the code and once both parties agree to the terms, the contract is executed automatically when the conditions are met.

Smart contracts are written in programming languages such as Solidity, which is used for the Ethereum blockchain, and can be deployed on blockchain platforms such as Ethereum, EOS, and TRON.

One of the key benefits of smart contracts is their ability to increase the efficiency and transparency of various types of transactions. They also have the potential to reduce the cost of transactions, as intermediaries are not needed to enforce the contract.

2.3. Ethereum

Ethereum is a decentralized, open-source blockchain platform that enables the creation of decentralized applications (dapps) and smart contracts. It was founded in 2014 by Vitalik Buterin, a programmer and cryptocurrency researcher.

Ethereum operates on a blockchain, which is a decentralized and distributed ledger technology. This allows for a more secure and transparent way of executing transactions and agreements compared to traditional centralized systems.

One of the key differences between Ethereum and other cryptocurrencies, such as Bitcoin, is that Ethereum was designed to be more than just a digital currency. Ethereum allows developers to build decentralized applications (dapps) on its platform, using its native programming language called Solidity. These dapps can run on the Ethereum network, which is maintained by a decentralized network of computers around the world.

Ethereum also has its own cryptocurrency, called Ether (ETH), which is used to pay for transactions on the Ethereum network and to compensate nodes for processing and verifying transactions.

Smart contracts are a key feature of Ethereum. They are self-executing contracts with the terms of the agreement between buyer and seller being directly written into code. Once both parties agree to the terms, the contract is executed automatically when the conditions are met, eliminating the need for intermediaries [9].

Ethereum also has a feature called gas, which is a unit of computation required to execute a contract or perform a transaction on the Ethereum network. The cost of gas is determined by the complexity of the computation and is paid in Ether.

2.3.1. Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) is the runtime environment for smart contracts on the Ethereum blockchain. It is responsible for executing the code of a smart contract and ensuring that it operates as intended.

The EVM is a sandboxed environment that runs on each node in the Ethereum network, allowing for the decentralized execution of smart contracts. It is responsible for executing the code of a smart contract, keeping track of the state of the contract, and ensuring that the terms of the contract are followed. Figure 2 shows components in an EVM machine.

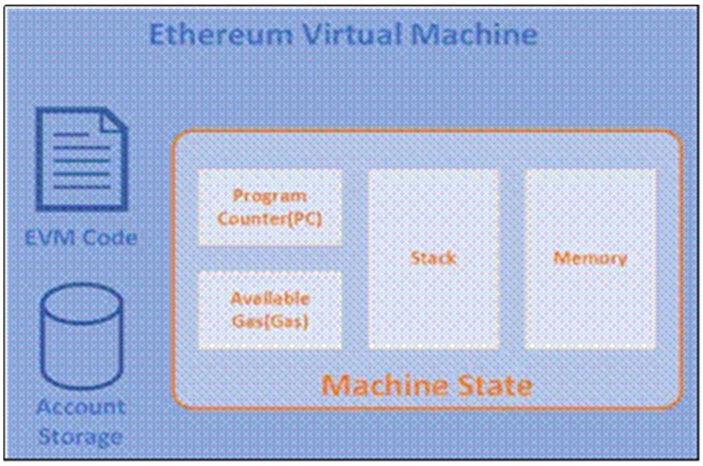


Figure 2 Ethereum Virtual Machine (EVM) [10]

The EVM is designed to be platform-agnostic and can run on any computer system, making it possible for anyone to participate in the Ethereum network. This helps to ensure the decentralized and secure execution of smart contracts on the Ethereum network [10].

One of the key benefits of the EVM is its ability to execute smart contracts in a secure and transparent manner, as all nodes in the Ethereum network can verify and execute the code. This makes it difficult for a single party to manipulate or alter the outcome of a contract, as all nodes must reach consensus before a contract can be executed [11].

2.3.2. Solidity

Solidity is the primary programming language used for writing smart contracts on the Ethereum platform. It is a high-level, contract-oriented programming language influenced by C++, Python, and JavaScript.

In Solidity, a smart contract is defined as a collection of code and data that resides on the Ethereum blockchain. Once deployed, a smart contract becomes a permanent part of the blockchain and can be executed by anyone on the network. The code and data within a smart contract can be interacted with through transactions submitted to the Ethereum network [12].

The code in a Solidity smart contract defines the rules and conditions for executing the contract, such as specifying the inputs and outputs, as well as the actions to be performed when the contract is executed. Solidity also includes features such as inheritance, libraries, and user-defined types, making it a powerful tool for creating complex decentralized applications [13].

When a transaction is submitted to the Ethereum network that interacts with a smart contract, the Ethereum virtual machine (EVM) executes the contract's code. The result of the execution is then recorded in the blockchain, making it transparent and tamper-proof. This allows for trustless transactions and eliminates the need for a middleman, making smart contracts an important building block for decentralized systems. Figure 3 demonstrate the flow process of development of a smart contract from developer to the EVM.

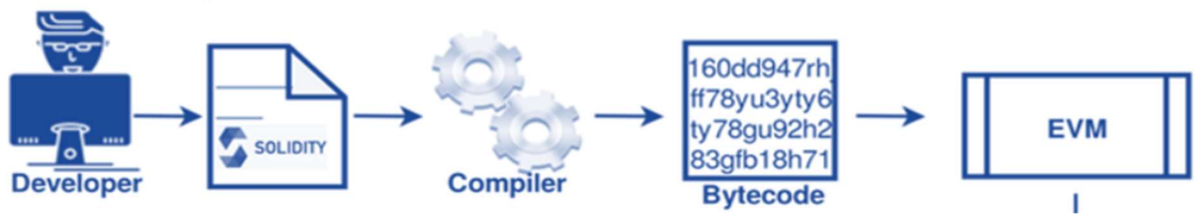


Figure 3 Development of Smart Contract [14]

2.3.3. Ethereum Smart Contract analysis and visualization

Ethereum smart contracts are self-executing contracts with the terms of the agreement directly written into code. They are stored and run on the Ethereum blockchain, making them transparent, tamper-proof, and decentralized.

To analyse and visualize Ethereum smart contracts, there are several tools available. Some popular tools are:

- **Remix**: A web-based IDE for writing, testing, and deploying smart contracts on the Ethereum network. It also includes a debugger and contract deployment options.
- **Etherscan**: A blockchain explorer for the Ethereum network that provides information about contracts, including the source code, transactions, and events.
- **Mythril**: A security analysis tool for Ethereum smart contracts that can identify potential vulnerabilities in your code.
- **DappHub**: An open-source platform for Ethereum development that provides a suite of developer tools, including a contract visualizer.
- **Alethio**: A blockchain data analytics platform that provides visualizations and insights into smart contracts and transactions on the Ethereum network.
- **Slither** is an open-source tool designed to analyse Ethereum smart contracts for security vulnerabilities and potential flaws. The tool is primarily used by developers and auditors to identify potential vulnerabilities in smart contracts before they are deployed to the Ethereum network.

  Slither works by analyzing the Solidity source code of a smart contract and identifying potential security issues such as potential reentrancy attacks, uninitialized storage, and other common vulnerabilities. The tool can also detect potential performance issues, such as inefficient code that could lead to high gas costs.
- **SmartCheck** uses symbolic execution, which means it analyzes the code paths of a smart contract to find potential security risks.

  SmartCheck can detect issues such as reentrancy, integer overflow/underflow, and other common security risks. It also supports a variety of other checks, including gas cost estimation and code complexity analysis.

  SmartCheck provides a web-based interface that allows users to upload their smart contract code for analysis. The tool generates an easy-to-read report that highlights any security issues found in the contract, along with recommendations for fixing them.
- **Honey Badger** uses static analysis techniques to analyze smart contracts without the need to execute them on the Ethereum network. Honey Badger can detect a variety of security issues, including reentrancy, integer overflow/underflow, and other common vulnerabilities. It also includes a feature that detects possible errors related to the transfer and storage of Ether. Honey Badger provides a command-line interface and can be used locally on a developer's machine. It generates a report that identifies any security vulnerabilities found in the contract, along with recommendations for fixing

them. One of the key advantages of Honey Badger is its ability to analyze large-scale contracts with many dependencies. It also supports the analysis of contracts written in Solidity, Vyper, and other Ethereum smart contract programming languages.

- **Smartbugs**; The tool analyzes the Solidity source code to detect various types of security issues, including common types of vulnerabilities such as integer overflow/underflow, reentrancy attacks, and unprotected functions, as well as some more specific ones related to smart contracts, like timestamp dependence, transaction-ordering dependence, and denial-of-service attacks [15]. SmartBugs uses a combination of static and dynamic analysis techniques to achieve its goal. It first performs a static analysis of the source code to identify potential issues, and then it generates test cases to execute the smart contract and verify the vulnerabilities. The dynamic analysis helps to confirm the potential issues detected by the static analysis and ensures that the smart contract behaves as expected under different conditions.
- **Manticore** is a symbolic execution tool designed to analyze smart contracts written in the Solidity programming language, which is used to create smart contracts on the Ethereum blockchain. Symbolic execution is a technique used in software testing that explores the possible paths of execution of a program by treating program inputs as symbolic expressions rather than concrete values. This allows for the exploration of all possible execution paths of a program, which is especially useful for finding bugs, vulnerabilities, and other security issues in smart contracts. Manticore uses symbolic execution to analyze smart contracts and can generate test cases and find vulnerabilities such as integer overflows, underflows, reentrancy bugs, and other issues that could lead to unexpected behavior in a smart contract. It also includes a fuzzing engine to test for security issues that may arise due to unexpected user input.
- **Solhint** is an open-source static analysis tool for Ethereum smart contracts written in Solidity. It helps developers to identify potential issues in their code before deployment, and ensures that the code conforms to best practices and industry standards. The tool can be integrated into various development environments such as Visual Studio Code, Sublime Text, and Atom, and can be used in continuous integration workflows to ensure code quality and security. Solhint is highly configurable and allows developers to customize the rules and settings to fit their specific needs. It is actively maintained and regularly updated with new rules and features to improve its functionality and effectiveness.

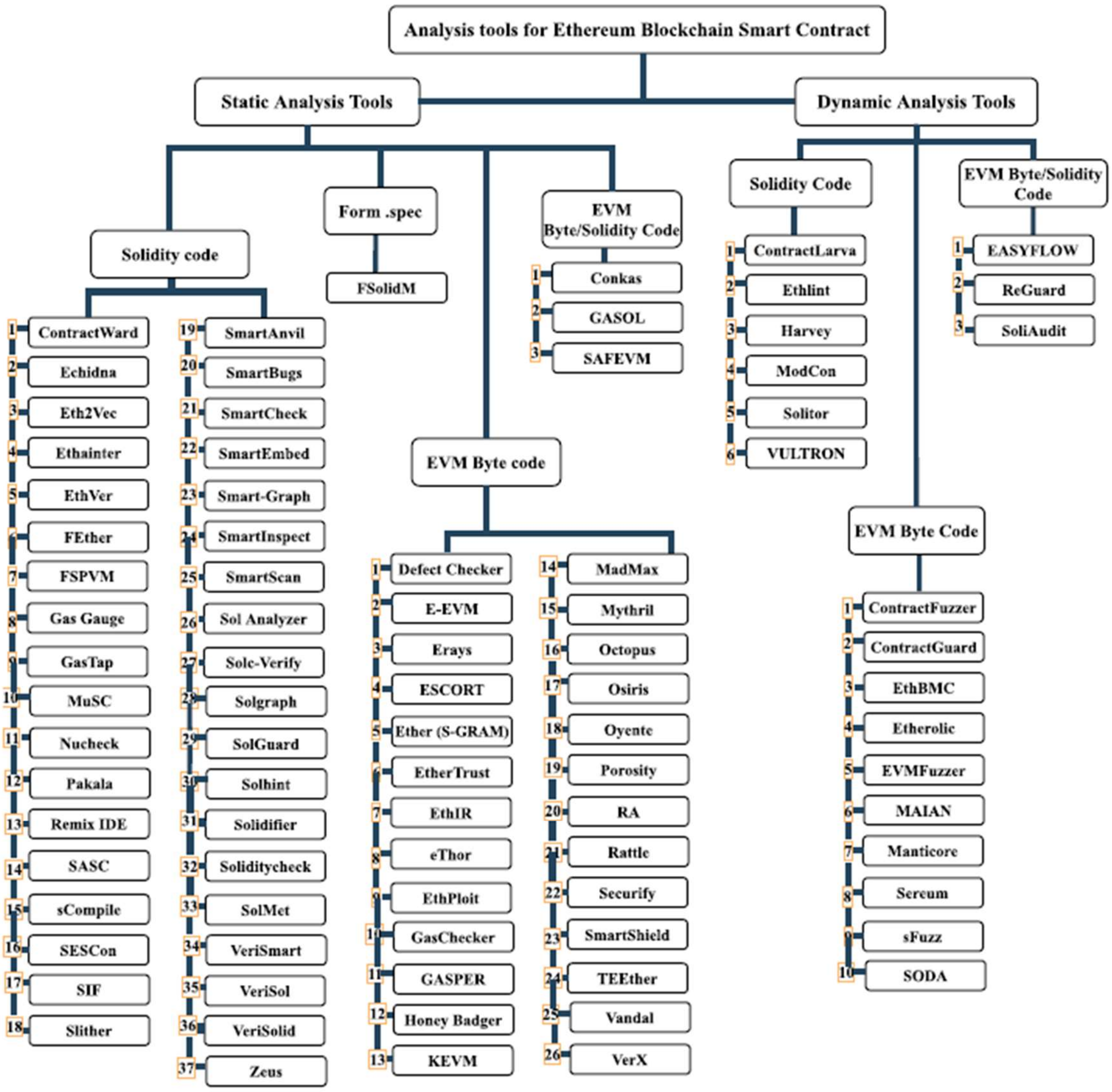


Figure 4 Ethereum Smart Contract Analysis Tools [16]

Figure 4 illustrates different Ethereum Smart Contract Analysis tools by categorizing them. These tools can help you to understand how a smart contract works, its transactions and events, and potential security risks. However, it's important to remember that even with these tools, you still need to have a good understanding of the underlying code and the Ethereum platform to properly analyse a contract.

2.3.4. Vulnerabilities in the smart contracts

Like any software, Ethereum Smart contracts can have vulnerabilities that could be exploited by attackers. Here are some of the common vulnerabilities in Ethereum smart contracts:

- **Reentrancy**: This occurs when a contract allows an attacker to repeatedly call it, resulting in unintended and potentially harmful behavior.
- **Unchecked-send**: This occurs when a contract fails to check the return value of a transfer or send operation, which can result in lost funds if the operation fails.
- **Integer overflow/underflow**: This occurs when an integer value exceeds the maximum or minimum value that can be stored in its data type, leading to unintended consequences.
- **Lack of access control**: This occurs when a contract fails to restrict access to its methods and data, allowing unauthorized actors to modify or extract sensitive information.
- **Unhandled exceptions**: This occurs when a contract does not properly handle exceptions that could occur during execution, resulting in unexpected and potentially harmful behavior.
- **Uninitialized storage pointers**: This occurs when a contract does not initialize storage pointers before use, which can lead to the use of uninitialized or invalid data.
- **Race conditions**: This occurs when two or more transactions attempt to modify the same state simultaneously, leading to unintended and potentially harmful behavior.
- **Timestamp dependence**: This occurs when a contract relies on the current timestamp, which can be manipulated by attackers, leading to unintended behavior.
- **Dos attacks**: This occurs when a contract is susceptible to denial-of-service (DoS) attacks, which can be used to congest the network and cause the contract to stop functioning.

These are just a few of the potential vulnerabilities in Ethereum smart contracts. It's important to thoroughly review and test contracts before deployment to reduce the risk of exploitation [16].

Reentrancy

A Reentrancy attack is a vulnerability that occurs when a contract allows an attacker to repeatedly call it, resulting in unintended and potentially harmful behaviour. Here's an example of a vulnerable code:

```
contract Attacker {
    address public victim;

    function attack() public {
        victim.call.value(1 ether)();
    }
}

contract Victim {
    uint public balance;

    function () public payable {
        balance += msg.value;
    }

    function withdraw() public {
        require(msg.sender.call.value(balance)());
        balance = 0;
    }
```

In this example, the Victim contract has a withdraw function that allows the owner to withdraw their balance. However, the withdraw function is vulnerable to reentrancy attacks because it calls msg.sender.call.value(balance)() before setting the balance to zero.

An attacker can create an instance of the Attacker contract and call its attack function, which in turn calls victim.call.value(1 ether)(). This will trigger the Victim contract's fallback function, causing its balance to be increased by 1 ether.

The withdraw function will then be executed, calling msg.sender.call.value(balance)() and sending the entire balance back to the attacker's address. This allows the attacker to repeatedly call attack and drain the Victim contract's balance.

To solve this issue, we can use a mutex (or a lock) to prevent reentrant calls. A mutex can be implemented using a flag variable that is set during execution and checked before entering a critical section of code. Here's an updated version of the Victim contract that is protected against reentrancy attacks:

```
contract Victim {
    uint public balance;
    bool public mutex = false;

    function () public payable {
        balance += msg.value;
    }

    function withdraw() public {
        require(!mutex);
        mutex = true;
        require(msg.sender.call.value(balance)());
        balance = 0;
        mutex = false;
    }
}
```

An **"unchecked-send"** attack refers to a security vulnerability in smart contracts that occurs when the contract executes a transfer of funds to an address without properly checking the validity of that address. This can result in funds being sent to an attacker's address instead of the intended recipient, leading to loss of funds.

Here's an example of a vulnerable Solidity code:

```
pragma solidity ^0.6.0;

contract Example {
    function transferFunds(address payable _to) public {
        _to.transfer(1000);
    }
}
```

In this example, the transferFunds function takes an address as an argument and transfers 1000 wei to that address. However, it does not check whether the address is a contract or not, or whether the contract has a fallback function that can steal the funds. As a result, if an attacker provides an address for a malicious contract that has a fallback function, the funds will be sent to the attacker's address.

To mitigate this issue, it is recommended to perform a check on the address before making the transfer, such as checking the balance of the address or verifying its code, to ensure that it is a valid address and not a malicious contract.Here's an example of a fixed code;

```
pragma solidity ^0.6.0;

contract Example {
    function transferFunds(address payable _to) public {
        require(_to.balance >= 1000, "Address does not have sufficient funds");
        _to.transfer(1000);
    }
}
```

In this example, the require statement checks whether the balance of the _to address is greater than or equal to 1000 wei before making the transfer. This helps to prevent the funds from being sent to an attacker's address.

Integer Overflow

Integer overflow and underflow are types of security vulnerabilities in software systems that arise from the handling of integer values. An integer overflow occurs when a calculation produces a value that is too large to be represented as an integer within the data type's storage size. An integer underflow occurs when a calculation produces a value that is too small to be represented within the data type's storage size.

Here's an example of code that is vulnerable to an integer overflow:

```
#include <stdio.h>

int main() {
  int x = 2147483647;  // maximum value for a 32-bit signed integer
  int y = 2147483647;
  int z = x + y;
  printf("The sum is %d\n", z);
  return 0;
}
```

In this example, the code calculates the sum of two maximum values for a 32-bit signed integer, which is 2147483647. The result of this calculation should be 4294967294, but because the data type int is limited to 32 bits, the result will actually be -2147483648. This is because the result overflows the int data type, and instead of being stored as the expected value, it wraps around to the minimum value that can be represented with 32 bits.

To prevent integer overflow, you can use a larger data type to store the result of the calculation, such as long long in C or int64_t in C++. Alternatively, you can perform the calculation in a way that checks for overflow before it occurs, for example, by using an intermediate value to store intermediate results.

Here's an updated version of the code that uses a larger data type to prevent the integer overflow:

```
#include <stdio.h>
#include <inttypes.h>

int main() {
  int32_t x = 2147483647;  // maximum value for a 32-bit signed integer
  int32_t y = 2147483647;
  int64_t z = (int64_t)x + (int64_t)y;
  printf("The sum is %" PRId64 "\n", z);
  return 0;
}
```

In this updated version of the code, the result of the calculation is stored in a 64-bit signed integer, which has a larger range of values and can therefore represent the result without overflow. The PRId64 macro is used with the printf function to print the value of z as a signed decimal integer.

Timestamp dependencies

Timestamp dependency attacks are a type of vulnerability in which an attacker exploits the relationship between a system's current time and the validity of certain data. For example, in some systems, a user's session may be considered valid only if the current time is within a specified range of the time that the session was established. If an attacker can manipulate the system's notion of the current time, they may be able to bypass authentication or gain unauthorized access.

Here's an example of code that is vulnerable to a timestamp dependency attack:

```
#include <time.h>
#include <stdio.h>

int main() {
  time_t now = time(NULL);
  time_t session_start = now - 60;  // 60 seconds ago
  time_t session_expiry = now + 60;  // 60 seconds in the future
  if (now >= session_start && now <= session_expiry) {
    printf("Session is valid\n");
  } else {
    printf("Session is invalid\n");
  }
  return 0;
}
```

In this example, the code calculates the current time using the time function, and then checks whether the current time is within a range of 60 seconds of the start time of a user session. If the current time is within this range, the code considers the session to be valid and allows the user to access protected resources.

However, an attacker could manipulate the system's notion of the current time to make it appear as though the current time is within the valid session range, even if the actual current time is outside of that range. For example, the attacker could set the system's clock back by 60 seconds to make it appear as though the session has not yet expired.

To prevent timestamp dependency attacks, it's important to use secure mechanisms for obtaining and storing the current time, and to validate timestamps in a way that is resistant to tampering. For example, you can use secure methods such as the Network Time Protocol (NTP) to obtain the current time from a trusted source, and you can store timestamps securely, such as in a secure database or using cryptographic signatures. Additionally, you can use cryptographic methods, such as secure hashing or encryption, to validate the integrity of timestamps and prevent tampering.

Lack of Access Control

Lack of access control is a type of vulnerability in which a system does not properly enforce authorization rules, allowing unauthorized access to protected resources. This can occur when a system does not properly check the permissions of a user before allowing access to a resource, or when a system does not enforce access controls consistently throughout the system.

Here's an example of code that is vulnerable to a lack of access control:

```
#include <stdio.h>

int main() {
  int user_role = 0;  // 0 = anonymous user, 1 = regular user, 2 = administrator
  int resource_id = 42;
  if (user_role == 2) {
    printf("Access granted to resource %d\n", resource_id);
  } else {
    printf("Access denied to resource %d\n", resource_id);
  }
  return 0;
}
```

In this example, the code checks the value of user_role to determine whether the user should be granted access to a protected resource with an ID of 42. If user_role is equal to 2, the code grants access to the resource and prints a message indicating that access has been granted. Otherwise, the code denies access and prints a message indicating that access has been denied.

However, this code is vulnerable to a lack of access control because the value of user_role can be easily manipulated by an attacker. For example, an attacker could modify the value of user_role to 2, effectively bypassing the access control check and gaining unauthorized access to the protected resource.

To prevent lack of access control vulnerabilities, it's important to implement proper access control checks throughout the system, and to validate the authenticity and integrity of any inputs used to enforce access control rules. Additionally, you can use role-based access control (RBAC) or attribute-based access control (ABAC) to enforce access control rules based on the attributes or roles of a user. For example, you can use a database or an external service to store the access control rules and roles for each user, and you can validate the user's role before granting access to a protected resource.

Denial of Service (DoS)

A Denial of Service (DoS) attack is a type of attack that aims to make a system or network resource unavailable to its intended users. This is often achieved by overwhelming the system with a large volume of requests, traffic, or data, causing it to become slow, unresponsive, or completely unavailable.

Here's an example of code that is vulnerable to a DoS attack:

```
#include <stdio.h>
#include <unistd.h>

int main() {
  while (1) {
    printf("Request received\n");
    sleep(1);  // simulate processing time
  }
  return 0;
}
```

In this example, the code runs an infinite loop that processes incoming requests and prints a message indicating that a request has been received. The sleep function is used to simulate processing time, but in a real-world scenario, this code would perform more complex and time-consuming operations.

However, this code is vulnerable to a DoS attack because an attacker could easily send an extremely high volume of requests to the system, causing it to become overwhelmed and slow or unresponsive. This could result in the system becoming unavailable to its intended users.

To prevent DoS attacks, it's important to implement rate limiting and other traffic management techniques to limit the volume of requests that a system can handle. For example, you can limit the number of requests that a system can process per second, and you can implement mechanisms to detect and block malicious traffic. Additionally, you can use caching and other performance optimization techniques to reduce the processing time required for each request, making it more difficult for an attacker to overwhelm the system.

2.3.5. Introduction to Artificial Intelligence as application to read smart contracts.

AI refers to the development of computer systems that can perform tasks that would typically require human intelligence, such as understanding natural language, recognizing objects, making decisions, and solving problems.

NLP (Natural Language Processing) is a subfield of AI that focuses on the interaction between computers and humans using natural language. NLP involves using machine learning algorithms to analyse, understand, and generate human language.

One application of NLP in AI is in reading vulnerabilities in Ethereum smart contracts. Ethereum is a blockchain platform that enables the creation and execution of decentralized applications known as smart contracts. These contracts are self-executing and can be programmed to perform various functions. However, like any other software, smart contracts can contain bugs and vulnerabilities that can be exploited by malicious actors [17].

An NLP-based AI system can be trained to analyse the code of smart contracts and identify potential security vulnerabilities. For example, the system could be trained on a large dataset of past vulnerabilities and exploitations in smart contracts, allowing it to detect similar patterns in new contracts. The AI system could also be trained to identify common coding patterns that are prone to bugs and vulnerabilities.

By using NLP and AI, organizations and individuals can perform automated security audits on their smart contracts, saving time and resources compared to manual audits. This can help ensure the security and integrity of Ethereum-based decentralized applications and prevent potential losses caused by exploitations [18].

2.3.6. Chat GPT API

ChatGPT is a state-of-the-art language generation model developed by OpenAI. It is trained on a massive dataset of diverse text sources and has the ability to generate human-like text based on the input provided to it.

Regarding the Ethereum smart contracts, the ChatGPT API could work as a tool for reading the information stored in these smart contracts. Ethereum is a decentralized platform that allows developers to build decentralized applications (dapps) and smart contracts. Smart contracts are self-executing code that run on the Ethereum blockchain and contain the terms of the agreement between the parties involved.

The ChatGPT API could be used to extract information from these smart contracts by interfacing with the Ethereum blockchain. The API could access the smart contract code and retrieve the relevant data, such as the terms of the agreement, the parties involved, and other important information stored in the contract.

Once the data has been retrieved, the ChatGPT API could then generate a human-readable summary or explanation of the information contained in the smart contract, making it easier for users to understand and use the information stored in the contract.

ChatGPT API could serve as a valuable tool for developers, organizations, and individuals looking to interact with Ethereum smart contracts and retrieve information from them in a user-friendly and accessible way.

2.3.7. Topic Modeling

Topic modeling can also be applied to the analysis of smart contracts, which are self-executing contracts with the terms of the agreement between buyer and seller being directly written into lines of code. In the context of smart contracts, topic modeling can be used to extract meaningful topics from the code and natural language descriptions of the contracts.

One application of topic modeling in smart contract analysis is to identify the key topics or themes covered by a large collection of smart contracts. This can help to understand the overall landscape of the smart contract ecosystem and the most common use cases for this technology.

Another application is to automatically classify smart contracts into different categories based on their topics. For example, topic modeling can be used to categorize smart contracts based on their intended purpose, such as financial contracts, voting contracts, or identity contracts.

Topic modeling can also be used to analyse the natural language descriptions of smart contracts, which can be written in plain English or other languages, to understand the human-readable provisions of the contracts. This information can be used to identify trends in the terms and conditions of smart contracts, or to help automate the process of summarizing the key provisions of a contract for a human reader [19].

2.3.8. Sentiment Analysis

Sentiment analysis is a subfield of natural language processing (NLP) that deals with identifying and extracting opinions and emotions from text data. In the context of Ethereum smart contracts, sentiment analysis can be applied to the natural language descriptions of the contracts to understand the overall sentiment expressed in the contracts [20].

For example, sentiment analysis can be used to determine whether a contract description is positive, negative, or neutral in tone. This information can be useful for several purposes, such as understanding the overall sentiment of the smart contract ecosystem, identifying contracts that are written in a contentious or controversial manner, or detecting potential risks associated with a contract.

Sentiment analysis can also be applied to the code of the smart contracts to identify any potential security risks or vulnerabilities in the code. For example, sentiment analysis can be used to identify code comments that express frustration or concern about a particular aspect of the code, which may indicate a potential security vulnerability.

Sentiment analysis can be performed using machine learning techniques, such as supervised learning and unsupervised learning, and various algorithms, such as Naive Bayes, Support Vector Machines (SVM), and Long Short-Term Memory (LSTM) neural networks. The choice of algorithm will depend on the specific requirements of the sentiment analysis task and the data being analysed [21].

Chapter 3

3. Related Works

Research in the past has explored the use of AI in smart contracts to improve their functionality and automate certain aspects of their execution. One area of research has focused on using machine learning algorithms to enable smart contracts to learn from past behaviour and make more informed decisions. Another area of research has looked at using AI to verify the correctness of smart contracts and ensure that they behave as intended. Additionally, researchers have explored the use of AI in creating more efficient and scalable smart contract systems. Some of the research papers are discussed as follows.

3.1. Research papers

Investigating the Role of Artificial Intelligence in Building Smart Contact on Block-Chain [22]. The research paper investigates the role of artificial intelligence (AI) in the development of smart contracts based on blockchain technology. The paper proposes using deep learning, a specific AI approach, to make smart contracts on the blockchain more efficient and intelligent.

The research paper takes into account existing studies in the areas of blockchain and AI but notes their limited scope and lack of accuracy and performance. The proposed work aims to provide a better solution in terms of accuracy and performance.

The paper seeks to explore how AI can enhance the functionality and usefulness of smart contracts in blockchain applications and provide a more robust and efficient approach to blockchain-based smart contract development. Figure 5 explains the workflow of a traditional smart contract.

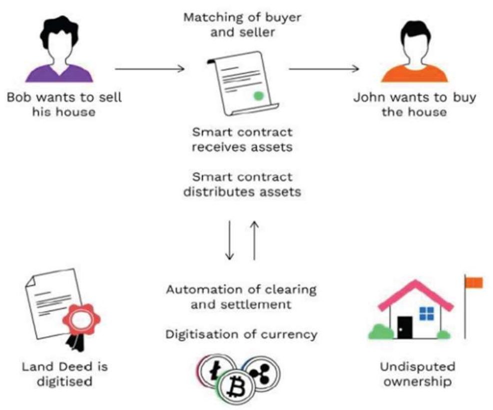


Figure 5 Traditional Smart Contract Workflow

Working of the proposed system; If certain criteria are met based on past experience, a smart contract will be activated. These contracts are automated, eliminating the need for a third party and streamlining the process for both parties. An AI module checks the transaction's reliability before using "if/when/then" instructions to build a smart contract on the blockchain. If past experience indicates potential issues, the smart contract will not be executed, saving time and resources. Figure 6 illustrates a proposed system of AI-based smart contract.

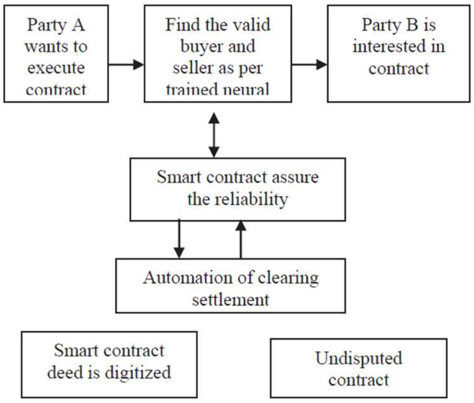


Figure 6 Proposed AI based Smart Contract

Smart Contract Privacy Protection Using AI in Cyber-Physical Systems: Tools, Techniques and Challenges [23]

The abstract discusses the increasing use of Blockchain (BC) technology and Cyber-Physical Systems (CPS), and the challenges associated with creating resilient and correct smart contracts (SCs) for these systems. SCs are important in modernizing traditional processes because they are self-executable, self-verifiable, and eliminate the need for trusted third-party systems, ultimately reducing administration and service costs, improving efficiency, and reducing security risks. However, SCs also present various security and privacy challenges that need to be addressed. The paper presents a survey on SC security vulnerabilities in software code that can be hacked by malicious users or compromise the entire BC network. The authors analyze various Artificial Intelligence (AI) techniques and tools for SC privacy protection and discuss open issues and challenges related to AI-based SCs. Finally, the paper presents a case study of retail marketing that uses AI and SCs to preserve its security and privacy.

Case Study Retail Marketing.

The specific scenario focuses on a retail marketing use case for an Ice cream Retailing Store (ICRS) based on blockchain (BC). Customers and ICRS are on the BC network, and customers can order ice cream within 5000m range and make payment using cash, UPI, credit/debit card, or internet banking. The ICRS promises delivery within 30 mins, or the full amount is refunded. The SC executes the workflow between the customer and ICRS, verifying order feasibility, and generating payment receipts.

To design and analyse complex SCs, AI algorithms are used to identify user preferences and offer seasonal discounts. Logic-based programming language is used to reduce the complexity of SCs, and AI uses dimensionality reduction techniques like principal component analysis to increase security by hiding sensitive customer information. AI-enabled SCs are more efficient and secure than traditional SCs. Figure 7 demonstrates a schematic diagram of the use case of retail marketing and how the AI based smart contract will work in the system.

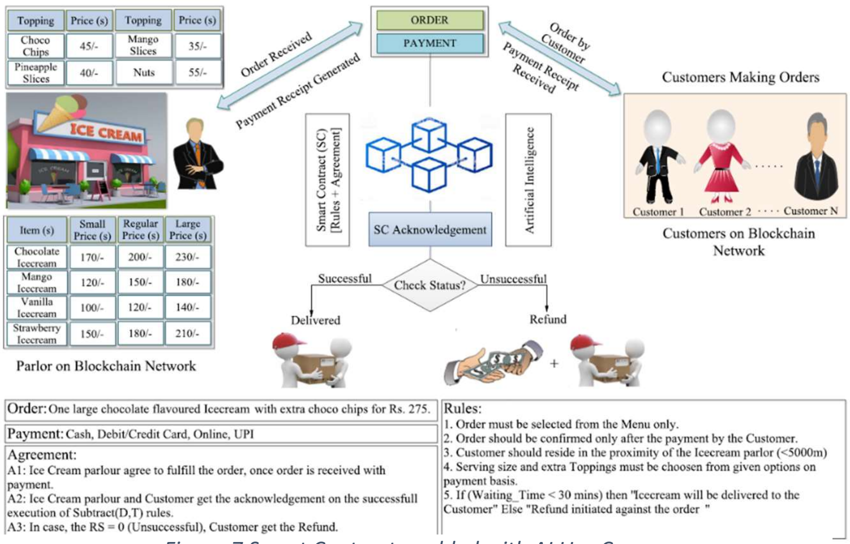


Figure 7 Smart Contract enabled with AI Use Case

Countermeasure Based on Smart Contracts and AI against DoS/DDoS Attack in 5G Circumstances [24]

The research paper discusses the vulnerability of servers to DoS/DDoS attacks in the context of the development of 5G. The increased data processing capability of 5G networks has not been matched by a corresponding enhancement in computer processing capability, which makes servers vulnerable to being compromised by attackers who send a massive amount of data. Further it proposes a solution that uses smart contracts and machine learning to protect servers from DoS/DDoS attacks by hiding them in a blockchain network and restricting the scale of the attacks via transaction fees. The system also uses non-repudiation of smart contracts to analyse users' behaviour and execute punishment for malicious communication. The proposed scheme is more effective than existing defence mechanisms in 4G networks and can counter DoS/DDoS attacks before they are launched. Furthermore, the proposed scheme avoids backdoors where model trainers could compromise AI models to launch DoS/DDoS attacks by considering the source trustworthiness of training samples.

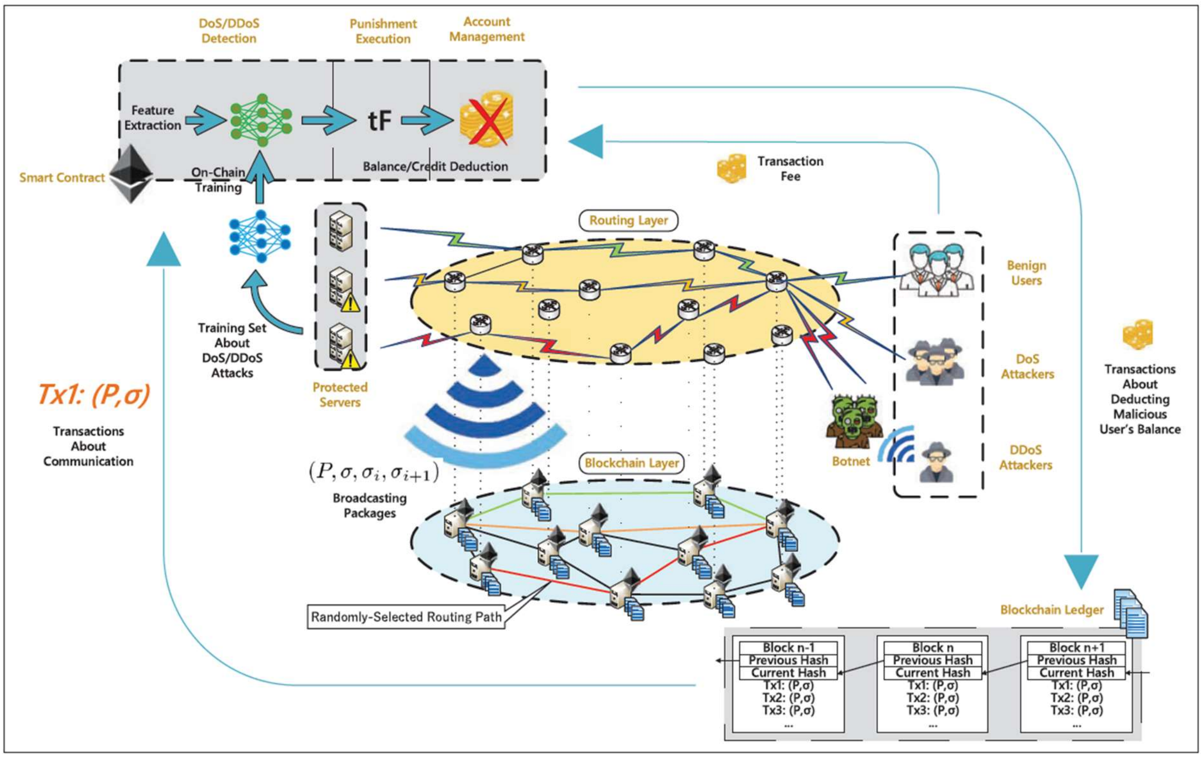


Figure 8 DoS/DDoS countermeasure overview

The Figure 8 describes a proposed defence framework against DoS/DDoS attacks that is based on AI-embedded smart contracts and runs on blockchain infrastructure. The framework consists of two main modules - the Routing module and Smart Contract module - both of which operate on the blockchain.

The protected server is hidden within the blockchain network, and external users interact with it through the blockchain. The Routing module is responsible for transferring external data by finding a route that connects external users and the server, while ensuring the integrity of data and preventing tampering by malicious nodes. The Routing module also broadcasts data packets to the blockchain to enable every blockchain node to extract and analyze the features of the data packets.

The Smart Contract module combines three modules: Account Management, DoS/DDoS Detection, and Punishment Execution. The Account Management module manages the accounts of external users, hosting users' records of balances and credits in the blockchain ledger. The DoS/DDoS Detection module trains the detection model to classify DoS/DDoS attacks and benign communication by analysing the features of the data and calculating the confidence coefficient to indicate the possibility of a DoS/DDoS attack. The Punishment Execution module calculates each transaction fee generated by miners' calculations, adjusts the fee rate of transactions based on the confidence coefficient provided by the DoS/DDoS Detection module, and then deducts the user balance and transfers it to the corresponding miner account.

The article proposes a solution to DoS/DDoS attacks in 5G circumstances using smart contracts and AI. The scheme adopts transaction fees in smart contracts to limit the scale of DoS/DDoS attacks and uses AI to calculate the confidence coefficient of attacks to increase the cost of launching them. The advantages of this scheme are that it embeds an AI module into smart contracts to ensure efficiency and trustworthiness, implements off-chain data routing with high reputation to ensure the efficiency of 5G communication and detection accuracy of attacks, and leverages the distributed infrastructure of the blockchain to prevent attackers from evading auditing of smart contracts. Compared to conventional schemes, this solution can counter attacks before they occur by amplifying the cost and forcing rational users not to launch attacks. The article suggests that this scheme could motivate new ideas for detecting DoS/DDoS attacks in 5G circumstances.

Comparison with the Previous Research Works

Compared to the previous research works, this thesis research uses new state of the art technology i.e. ChatGPT for analysing Ethereum smart contracts, which is a new and innovative approach to this field. Previous research has mainly relied on manual analysis techniques or more traditional data analysis tools, which can be time-consuming and less accurate.

The research utilizes advanced NLP techniques for analysing the text data extracted from smart contracts. Previous research has primarily focused on analysing the code and structure of smart contracts, rather than the text data contained within them. This approach allows for a more comprehensive analysis of smart contracts and can provide insights into the emotions and opinions expressed in the text, which can be crucial for understanding the risks associated with smart contracts.

The research emphasizes the accessibility and understanding of smart contracts for individuals who may not have technical knowledge or expertise. The use of the ChatGPT API for automated analysis of smart contracts significantly reduces the time and effort required to review and analyse them, making it easier for individuals with little to no technical background to understand the risks associated with smart contracts. This approach can also help to increase transparency and trust in the use of smart contracts in various industries.

Chapter 4

4. Realized System

In this part of the thesis, it will be discussed; the proposed system that caters the problems associated in reading Ethereum smart contracts and the use of AI techniques to form our results.

4.1. Proposed System

To break it down the proposed system, here's how the system would work:

1. The system uses the ChatGPT-3.5 (Davinci-003) to read Ethereum smart contracts. ChatGPT is a large language model that can be used for a wide range of natural language processing tasks, including reading text.
2. A python script is developed that uses ChatGPT API to analyse the smart contracts for weaknesses and vulnerabilities, results are stored in a CSV file.
3. With the results stored in the CSV file, then topic modeling analysis was performed. Topic modeling is a technique for identifying topics or themes in a set of documents (in this case, the smart contract weakness results). This could help identify patterns or trends in the types of weaknesses that are present in the contracts.
4. Finally, sentiment analysis on the results was performed. Sentiment analysis is a technique for identifying the emotional tone of a piece of text. In this case, we used sentiment analysis to determine whether the results are generally positive (i.e., few weaknesses), negative (i.e., many weaknesses), or neutral.

This proposed system would allow to analyse Ethereum smart contracts for potential weaknesses and then gain insights into the patterns and trends present in the weaknesses that are identified. This could be useful for identifying areas where improvements could be made in smart contract development and for monitoring the security of smart contracts over time.

Figure 9 illustrates the workflow of the proposed system for this thesis.

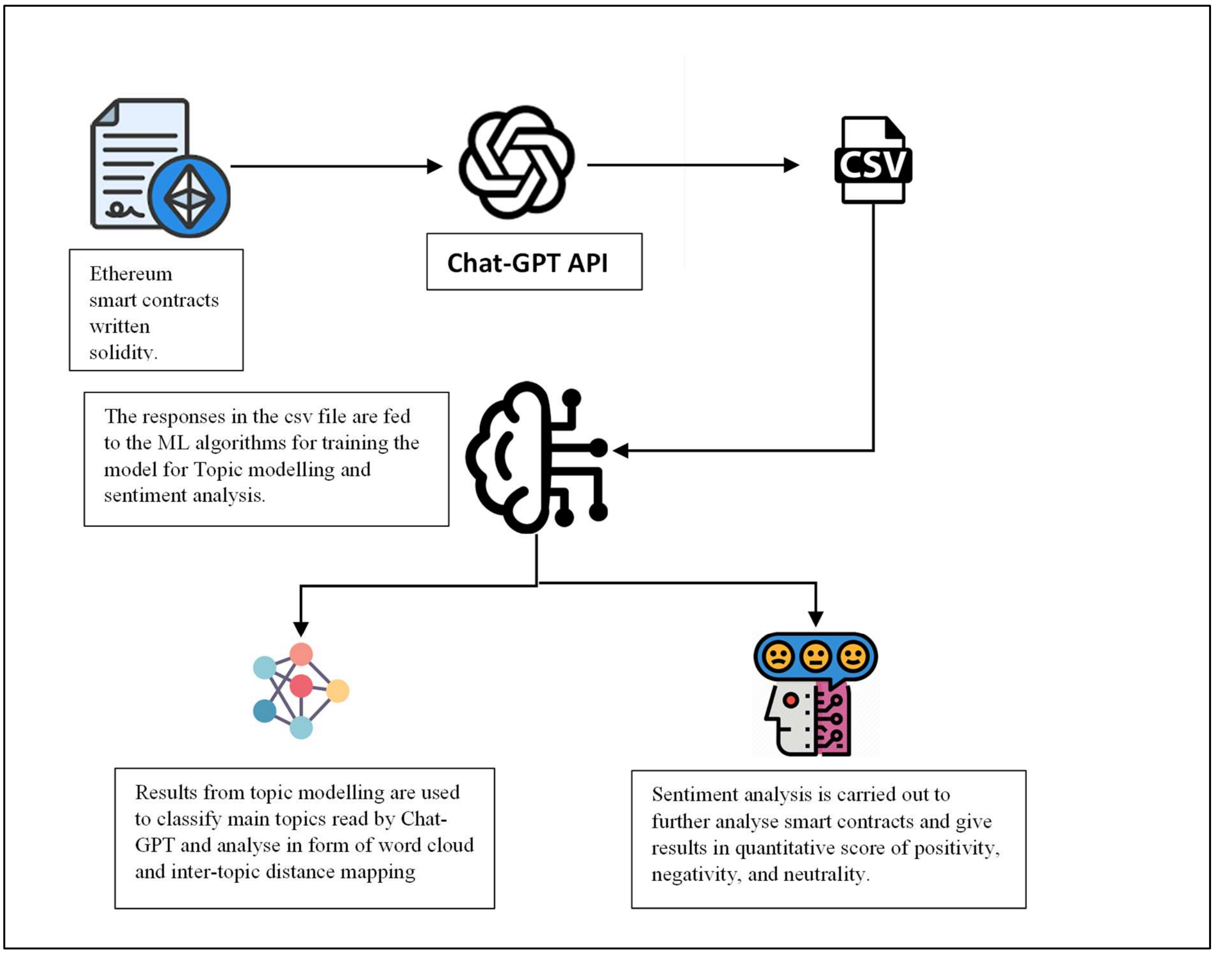


Figure 9 Process Flow of our proposed system

4.2. Dataset of smart contracts

Dataset of Ethereum smart contracts from a repository [25] is used in the thesis. The repository consists of 40,000 smart contracts which has about 4 types of vulnerabilities in them. The author developed a python script for testing each type of vulnerability individually, however in this thesis the dataset was downloaded and stored in a folder which was further read by a python script that uses Chat-GPT API to analyse all types of vulnerabilities in the smart contract in a single batch process. Overall, 5000 smart contracts were analysed using Chat-GPT API from which approximately 3000 smart contracts were read properly, and results acquired due to some limitation in Chat-GPT.

The Annotated Dataset

The annotated dataset is classified by the types of vulnerabilities found the in smart contracts. The classified vulnerabilities in the dataset were Delegate call, Reentrancy, Integer Overflow and Timestamp.

- The total smart contracts which had issue of Delegate call were accounted of total 196 contracts.
- The total smart contracts which had issue of Integer Overflow were accounted of total 275 contracts.
- The total smart contracts which had issue of Reentrancy were accounted of total 273 contracts.
- The total smart contracts which had issue of Timestamps were accounted of total 349 contracts.

4.3. Use of Chat-GPT API

A python script is developed that uses Chat-GPT API to read Ethereum smart contracts.

```
import os
import openai
import csv

openai.api_key = "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"
path='C:/Users/39349/Desktop/contract_dataset_ethereum/contract5/'
file_names= os.listdir(path)

ask="what is the weaknesses in the following smart contract solidity
code?\n\n"

if not file_names:
    print("Directory is Empty")
else:
    with open("resultscontract5.csv", "w", newline="") as f:
        writer = csv.writer(f)
        writer.writerow(["File Name", "ChatGPT Response"])
        for i in file_names:
            try:
                print('\nChatGPT Response for file '+i)
                with open(path+i) as my_file:
```

```
                response = openai.Completion.create(
                    model="text-davinci-003",
                    prompt=ask+ my_file.read(),
                    temperature=0.9,
                    max_tokens=2048,
                    top_p=1,
                    frequency_penalty=0,
                    presence_penalty=0.6,
                    stop=[" Human:", " AI:"]
                )
                text = response['choices'][0]['text']
                print (text)
                writer.writerow([i, text])
        except:
            pass
```

This code is written in Python and its purpose is to use OpenAI's API to generate responses from a language model called "text-davinci-003". The goal is to analyze a set of smart contract solidity code files and generate a response for each file that highlights any weaknesses or vulnerabilities in the code.

The first three lines of the code import three Python modules: os, openai, and csv. The os module provides a way to interact with the operating system, the openai module provides an interface to the OpenAI API, and the csv module is used for reading and writing CSV files.

The fourth line sets the API key for the OpenAI API, which is required to access the service. The next line defines the path to the directory where the smart contract solidity code files are located. The os.listdir function is then used to create a list of the names of all the files in the directory.

The next line defines a string variable called ask, which is used as the prompt for the language model. The prompt is the initial text that is given to the model as input, and it is customized in this case to ask for weaknesses in the code.

The if not file_names statement checks if the list of file names is empty, which would indicate that there are no files in the directory. If the directory is empty, the program prints a message indicating this. If the directory is not empty, the program proceeds to analyze each file.

The with open statement creates a new CSV file called "resultscontract5.csv" and defines the headers for the file as "File Name" and "ChatGPT Response". The for loop iterates over each file in the directory, and the try block attempts to generate a response for each file using the OpenAI API.

Inside the try block, the open function is used to open the file and read its contents. The openai.Completion.create function is then used to generate a response from the language model, which is based on the prompt ask concatenated with the contents of the file. The function is called with several arguments to specify various parameters for the response, such as the temperature, max tokens, and penalties.

The response from the language model is stored in the text variable, and it is printed to the console. The response is also written to the CSV file using the writer.writerow function.

The except block is used to handle any exceptions that may occur during the analysis of a file, such as a file not being found or an error with the OpenAI API. If an exception occurs, the program simply skips the file and continues to the next one.

This code is a useful example of how to use OpenAI's API to generate responses from a language model, and it demonstrates how this can be used to analyze a set of files in a specific domain.

4.4. Results from Chat-GPT

The below table shows some of the results extracted from Chat-GPT

Smart Contracts number	Chat-GPT responses
101	The weaknesses in the above smart contract Solidity code are: 1. There is no check for totalSupply against the maximum of 2^256 - 1. 2. The function approveAndCall is potentially vulnerable to a reentrancy attack. 3. The balances are updated before fundsWallet.transfer(msg.value) is called, which could result in the funds being lost if the withdrawal fails.

102	1. The contract does not include any protection from replay attacks. 2. No authorization is checked before setting or getting data, anyone could change the reference or get the reference value. 3. No validation for the reference values are present. 4. There is no mechanism to safely delete or update the data stored in the audit log. 5. There is no method to check the integrity of the stored data since there is no hash or signature verification.
104	1. The contract does not include any safety measures to prevent the Gastoken contract from being abused. 2. The contract does not check that the _merkleMineContract address, _recipients, and _merkleProofs are valid before proceeding. 3. There is no limit on the amount of tokens that can be burned and freed. 4. There is no enforcement of any specific standards when approving tokens to be freed and/or transferred. 5. The Livepeer address is hardcoded, and could potentially be exploitable.
107	1. There is no restriction on how much ERC20 tokens can be minted, allowing for inflation. 2. No safety mechanism is built into the contract to check for overflow/underflow when subtracting or adding values. 3. No way to freeze accounts

	4. There is no way to ensure funds are used properly as there is no audit trail of the transactions. 5. The transfer() and transferFrom() functions are not restricted to a certain amount of tokens, allowing for unlimited transfers.
113	Some of the weaknesses in this contract: 1. The total supply is hardcoded. 2. The new owner address is stored in a publicly visible variable which could be read and changed by attackers. 3. No function to mint more tokens has been defined which means the total supply will remain fixed forever. 4. No functionality for pausing or freezing accounts has been defined.

Table 1 Results from Chatgpt

Table 1 illustrates results from ChatGPT of few smart contracts that were analysed out of total of 2500 approx smart contracts that were read and the results were saved in a csv file. 17 smart contracts had no vulnerabilities, however ChatGPT suggested some improvements in almost every smart contract.

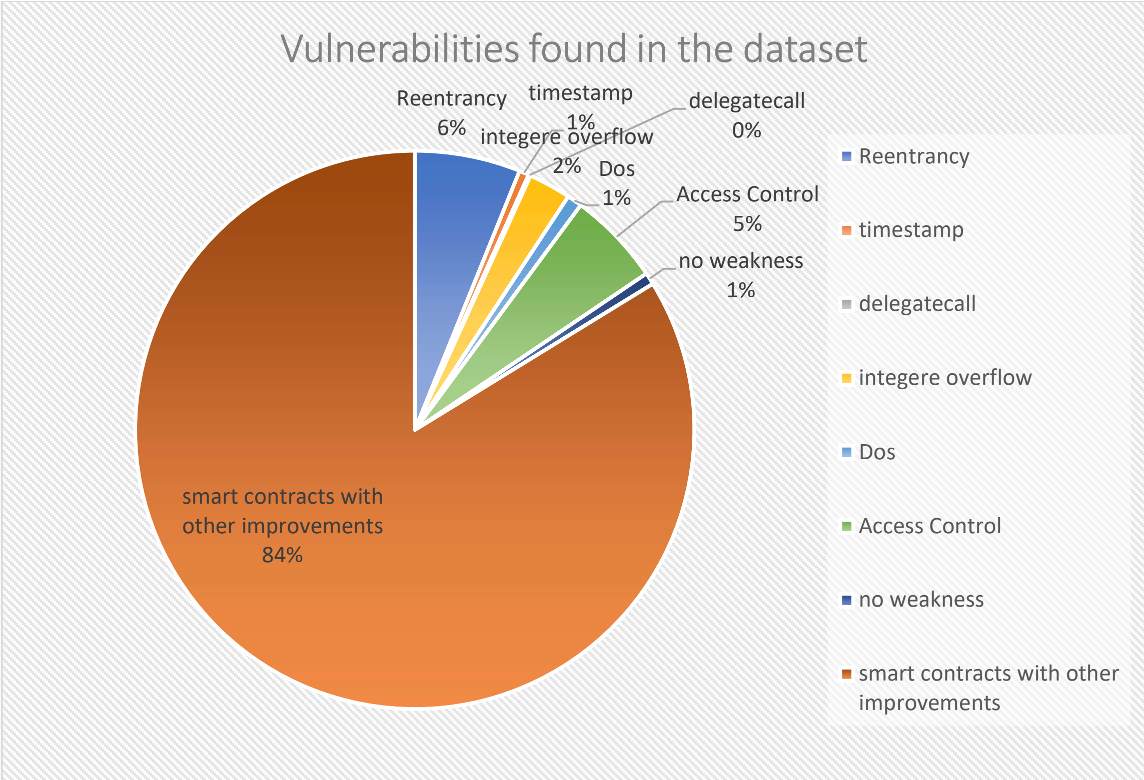


Figure 10 Vulnerabilities found in the dataset

Figure 10 shows the percentages of the major vulnerabilities found in the smart contract dataset analysed by ChatGPT, About 17 contracts were free of any vulnerability. The main vulnerability was found to be the attacks of Reentrancy 6% and Lack of access control 5%.. However, ChatGPT proposed some kind of improvements in the code of 84% of the remaining smart contracts.

4.5. Topic Modelling

In this thesis we used Latent Dirichlet Allocation (LDA) that is a generative statistical model used for topic modeling. The model represents each document as a mixture of topics and each topic as a distribution over words. In other words, LDA is a way to automatically discover topics that are present in a collection of documents.

Gensim was used; a popular open-source Python library for topic modeling and natural language processing. It provides an implementation of LDA and other algorithms for topic modeling [27].

Here is the breakdown of the code used;

The code is a Python script that reads a CSV file, performs text pre-processing on the data, and trains an LDA (Latent Dirichlet Allocation) model using Gensim.

Specifically, the code does the following:

```
'''
Load the dataset from the CSV and save it to 'data_text'
'''
import pandas as pd
data = pd.read_csv('/content/combined.csv',encoding='latin-1', error_bad_lines=False);
print(data.head())
```

```
[ ] # We only need the Headlines text column from the data
    data_text = data[['Responses']];
    data_text['index'] = data_text.index

    documents = data_text
```

```
[ ] '''
    Loading Gensim and nltk libraries
    '''
    # pip install gensim
    import gensim
    from gensim.utils import simple_preprocess
    from gensim.parsing.preprocessing import STOPWORDS
    from nltk.stem import WordNetLemmatizer, SnowballStemmer
    from nltk.stem.porter import *
    import numpy as np
    np.random.seed(400)
```

```
[ ] import nltk
    nltk.download('wordnet')

    [nltk_data] Downloading package wordnet to /root/nltk_data...
    True
```

```
[ ] import nltk
    nltk.download('omw-1.4')
    print(WordNetLemmatizer().lemmatize('went', pos = 'v')) # past tense to present tense

    [nltk_data] Downloading package omw-1.4 to /root/nltk_data...
    go
```

It imports the necessary libraries, such as Pandas, Gensim, and NLTK, and loads the dataset from a CSV file using Pandas.

```
[ ] stemmer = SnowballStemmer("english")
    original_words = ['caresses', 'flies', 'dies', 'mules', 'denied','died', 'agreed', 'owned',
               'humbled', 'sized','meeting', 'stating', 'siezing', 'itemization','sensational',
               'traditional', 'reference', 'colonizer','plotted']
    singles = [stemmer.stem(plural) for plural in original_words]

    pd.DataFrame(data={'original word':original_words, 'stemmed':singles })
```

[ ]

	original word	stemmed
0	caresses	caress
1	flies	fli
2	dies	die
3	mules	mule
4	denied	deni
5	died	die
6	agreed	agre
7	owned	own
8	humbled	humbl
9	sized	size
10	meeting	meet
11	stating	state
12	siezing	siez
13	itemization	item
14	sensational	sensat
15	traditional	tradit
16	reference	refer
17	colonizer	colon

```
[ ] '''
    Write a function to perform the pre processing steps on the entire dataset
    '''
    def lemmatize_stemming(text):
        return stemmer.stem(WordNetLemmatizer().lemmatize(text, pos='v'))

    # Tokenize and lemmatize
    def preprocess(text):
        result=[]
        for token in gensim.utils.simple_preprocess(text) :
            if token not in gensim.parsing.preprocessing.STOPWORDS and len(token) > 3:
                # TODO: Apply lemmatize_stemming() on the token, then add to the results list
                result.append(lemmatize_stemming(token))

        return result
```

```
[ ] '''
    Preview a document after preprocessing
    '''
    document_num = 551
    doc_sample = documents[documents['index'] == document_num].values[0][0]

    doc_sample
```

```python
print("Original document: ")
words = []
for word in doc_sample.split(' '):
    words.append(word)
print(words)
print("\n\nTokenized and lemmatized document: ")
print(preprocess(doc_sample))
```

```python
doc_sample = 'the doctor came to the village to visit some patients yesterday.'
print("Original document: ")
words = []
for word in doc_sample.split(' '):
    words.append(word)
print(words)
print("\n\nTokenized and lemmatized document: ")
print(preprocess(doc_sample))
```

```
Original document:
['the', 'doctor', 'came', 'to', 'the', 'village', 'to', 'visit', 'some', 'patients', 'yesterday.']


Tokenized and lemmatized document:
['doctor', 'come', 'villag', 'visit', 'patient', 'yesterday']
```

It pre-processes the text data by tokenizing the text, removing stop words, and lemmatizing the remaining words.

```python
[ ] # TODO: preprocess all the headlines, saving the list of results as 'processed_docs'

    processed_docs = documents['Responses'].astype(str).map(lambda x: preprocess(x))
```

```python
'''
Preview 'processed_docs'
'''
processed_docs[:10]
```

```
0    [perform, fast, snapdragon, display, great, sc...
1    [yesterday, come, navi, blue, look, great, rep...
2    [perform, fast, snapdragon, display, great, sc...
3    [perform, fast, snapdragon, display, great, sc...
4    [perform, fast, snapdragon, display, great, sc...
5    [purchas, list, model, receiv, make, feel, lur...
6    [receiv, today, switch, origin, pixel, wasnt, ...
7    [purchas, list, model, receiv, make, feel, lur...
8    [purchas, list, model, receiv, make, feel, lur...
9    [purchas, list, model, receiv, make, feel, lur...
Name: TEXT, dtype: object
```

```
[ ] '''
    Create a dictionary from 'processed_docs' containing the number of times a word appears
    in the training set using gensim.corpora.Dictionary and call it 'dictionary'
    '''
    dictionary = gensim.corpora.Dictionary(processed_docs)
```

```
[ ] '''
    Checking dictionary created
    '''
    count = 0
    for k, v in dictionary.iteritems():
        print(k, v)
        count += 1
        if count > 10:
            break
```

```
0 add
1 address
2 array
3 check
4 code
5 curconfig
6 game
7 limit
8 meet
9 requir
10 sender
```

It creates a dictionary from the pre-processed text data using Gensim's Dictionary class.

```
[ ] '''
    Create the Bag-of-words model for each document i.e for each document we create a dictionary reporting how many
    words and how many times those words appear. Save this to 'bow_corpus'
    '''
    # TODO
    bow_corpus = [dictionary.doc2bow(doc) for doc in processed_docs]
```

```
[ ] '''
    Checking Bag of Words corpus for our sample document --> (token_id, token_count)
    '''
    bow_corpus[document_num]
```

```
[ ] '''
    Preview BOW for our sample preprocessed document
    '''
    # Here document_num is document number 4310 which we have checked in Step 2
    bow_doc_4310 = bow_corpus[document_num]

    for i in range(len(bow_doc_4310)):
        print("Word {} (\"{}\") appears {} time.".format(bow_doc_4310[i][0],
                                                         dictionary[bow_doc_4310[i][0]],
                                                         bow_doc_4310[i][1]))
```

```
Word 19 ("unlimit") appears 1 time.
Word 24 ("maximum") appears 1 time.
Word 34 ("method") appears 3 time.
Word 42 ("burn") appears 1 time.
Word 43 ("enforc") appears 1 time.
Word 49 ("freez") appears 1 time.
Word 70 ("correct") appears 1 time.
Word 72 ("decim") appears 1 time.
Word 82 ("cod") appears 1 time.
Word 86 ("hard") appears 1 time.
Word 127 ("burnfrom") appears 1 time.
Word 209 ("permiss") appears 1 time.
Word 300 ("minimum") appears 1 time.
Word 309 ("want") appears 1 time.
Word 310 ("default") appears 1 time.
Word 313 ("block") appears 1 time.
```

It creates a bag-of-words model for each document, where the bag-of-words model reports how many words and how many times those words appear.

.

```
[ ] '''
    Create tf-idf model object using models.TfidfModel on 'bow_corpus' and save it to 'tfidf'
    '''
    from gensim import corpora, models

    # TODO
    tfidf = models.TfidfModel(bow_corpus)
```

```
[ ] '''
    Apply transformation to the entire corpus and call it 'corpus_tfidf'
    '''
    # TODO
    corpus_tfidf = tfidf[bow_corpus]
```

```python
'''
Preview TF-IDF scores for our first document --> --> (token_id, tfidf score)
'''
from pprint import pprint
for doc in corpus_tfidf:
    pprint(doc)
    break
```

```
[(0, 0.5555142153693671),
 (1, 0.6501999292412817),
 (2, 0.4313729920151899),
 (3, 0.2873349096386506)]
```

```python
# LDA mono-core -- fallback code in case LdaMulticore throws an error on your machine
# lda_model = gensim.models.LdaModel(bow_corpus,
#                                    num_topics = 10,
#                                    id2word = dictionary,
#                                    passes = 50)

# LDA multicore
'''
Train your lda model using gensim.models.LdaMulticore and save it to 'lda_model'
'''
# TODO
lda_model = gensim.models.LdaMulticore(bow_corpus, num_topics = 10, id2word = dictionary, passes = 150)
```

```python
'''
For each topic, we will explore the words occuring in that topic and its relative weight
'''
for idx, topic in lda_model.print_topics(-1):
    print("Topic: {} \nWords: {}".format(idx, topic))
    print("\n")
```

It creates a Tf-idf (term frequency-inverse document frequency) model object using Gensim's TfidfModel class and applies the transformation to the entire corpus

Further, It trains an LDA model using Gensim's LdaMulticore class on the tf-idf transformed corpus.

Overall, the code demonstrates how to perform text preprocessing and train an LDA model using Gensim.

4.6. Sentiment Analysis

Further sentiment analysis was performed on the responses from Chat-GPT. This code performs topic modeling and sentiment analysis on a CSV file containing text data. Here's a brief explanation of what each part of the code does:

- Import necessary libraries

```python
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import LatentDirichletAllocation
```

```
from nltk.sentiment import SentimentIntensityAnalyzer
import csv
```

This code imports the necessary libraries for performing topic modeling and sentiment analysis.

- Load the data from the CSV file

```
data =
pd.read_csv('C:/Users/39349/PycharmProjects/pythonProject2/combined.csv'
,encoding='latin-1', on_bad_lines='skip', engine='python')
data = data.dropna()
text = data['Responses']
```

This code loads the CSV file using pandas, drops any rows that contain missing data, and assigns the column 'Responses' to the variable text.

- Perform topic modeling using Latent Dirichlet Allocation

```
vectorizer = CountVectorizer(max_df=0.95, min_df=2,
stop_words='english')
doc_term_matrix = vectorizer.fit_transform(text)
lda = LatentDirichletAllocation(n_components=10, random_state=42)
lda.fit(doc_term_matrix)
```

This code performs topic modeling using Latent Dirichlet Allocation. It uses the CountVectorizer class to create a document-term matrix from the text data, and then fits an LDA model to the matrix.

- Extract the topics and their corresponding weights

```
topics = lda.components_
topic_weights = lda.transform(doc_term_matrix)
```

This code extracts the topics and their corresponding weights from the LDA model.

- Perform sentiment analysis using the NLTK library

```
sia = SentimentIntensityAnalyzer()
sentiments = []
```

```
for doc in text:
    sentiments.append(sia.polarity_scores(doc))
```

This code performs sentiment analysis using the SentimentIntensityAnalyzer class from the NLTK library. It creates a list of sentiment scores for each document in the text data.

- Print the topics and sentiments.

```
for i, topic in enumerate(topics):
    print("Topic ", i, ": ",
[(vectorizer.get_feature_names_out()[index], weight) for index, weight
in enumerate(topic)])
```

- Write the topics and sentiments to a CSV file

```
with open('topics_sentiments3.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(["Topic", "Words", "Sentiments"])
    for i, topic in enumerate(topics):
        for index, weight in enumerate(topic):
            writer.writerow([i,
vectorizer.get_feature_names_out()[index], weight])
    for sentiment in sentiments:
        writer.writerow(["Sentiment", sentiment])
```

This code writes the topics and sentiments to a CSV file. It first opens the file using the open function and creates a csv.writer object. It then writes the topic and word weights to the file, followed by the sentiment scores for each document.

This code performs topic modeling and sentiment analysis on a CSV file containing text data, and produces the following results:

- Topic modeling using Latent Dirichlet Allocation (LDA): The code uses LDA to extract 10 topics from the text data, along with their corresponding weights. For each topic, the code prints the top words with the highest weights in that topic.
- Sentiment analysis using the NLTK library: The code uses the SentimentIntensityAnalyzer class from the NLTK library to perform sentiment analysis on each document in the text data. The code creates a list of sentiment scores for each

document, including the polarity scores for positive, negative, and neutral sentiment, as well as the compound score, which represents an overall score for the sentiment of the text.

- Writing the topics and sentiments to a CSV file: The code writes the topics and their corresponding words and weights, as well as the sentiment scores for each document, to a CSV file named 'topics_sentiments3.csv'. The file has three columns: 'Topic', 'Words', and 'Sentiments'. The 'Topic' column contains the topic number (0-9), the 'Words' column contains the top words with the highest weights in the topic, and the 'Sentiments' column contains the sentiment scores for each document.
- this code allows for the exploration of the main topics present in the text data and the sentiment associated with each document. This information can be useful for understanding the themes and emotions present in the data and can inform further analysis or decision-making.

Chapter 5

5. Evaluation

5.1. Word Cloud from Topic Modeling

Word clouds are a visual representation of text data in which the size of each word is proportional to its frequency in the text. They are often used to highlight the most important or frequent words in a large corpus of text, and can be a useful tool for exploring and summarizing textual data.

```
[1] import matplotlib.pyplot as plt
    %matplotlib inline

    from wordcloud import WordCloud, STOPWORDS
```

```
def word_cloud(topic, model):
    plt.figure(figsize = (8,6))
    topic_words = [model.print_topic(topic, 75)]
    cloud = WordCloud(stopwords = STOPWORDS, background_color = 'white',
                      width=2500, height=1800).generate(" ".join(topic_words))

    print('\nWordcloud for topic:', topic, '\n')
    plt.imshow(cloud)
    plt.axis('off')
    plt.show()
```

```
for topic in range(10):
    #plt.figure(figsize=(10,15))
    word_cloud(topic, lda_model)
```

This code is used to generate word clouds for topics in a topic modeling analysis, specifically using the Latent Dirichlet Allocation (LDA) algorithm.

First, the necessary modules are imported: matplotlib.pyplot for visualizations and %matplotlib inline for inline plotting in Jupyter notebooks; and WordCloud and STOPWORDS from the wordcloud module for generating the word clouds.

Then, a function named word_cloud is defined with two arguments: topic and model. The topic argument is the index number of the topic to generate the word cloud for, and the model argument is the LDA model object.

The function creates a new figure with a specific size, generates a list of the top 75 words for the specified topic using the LDA model's print_topic method, and then creates a word cloud from those words using the WordCloud function. The STOPWORDS argument is used to remove common English words from the word cloud. Finally, the function displays the word cloud using imshow and show functions from matplotlib.pyplot.

The for loop iterates over 10 topics, calling the word_cloud function for each topic to generate a corresponding word cloud.

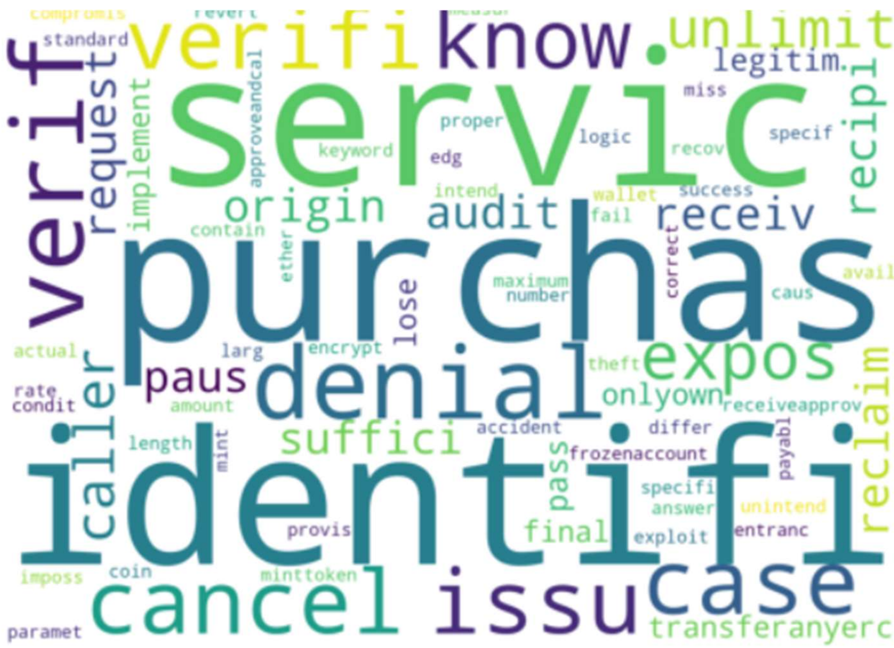
5.2. Results from Topic Modeling

- Word cloud for Topic 0

- Word Cloud for Topic 1

- Word cloud for Topic 2

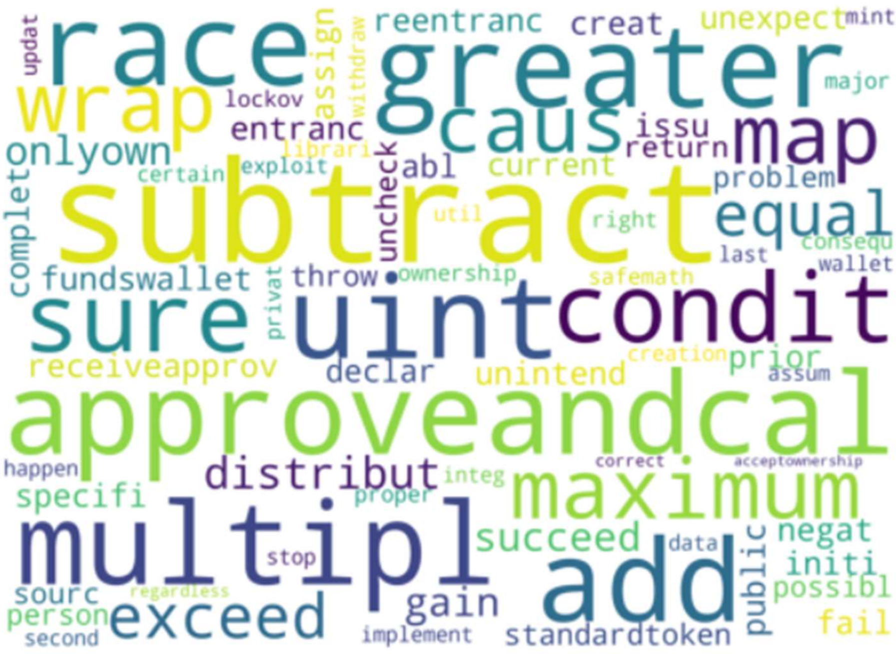
- Word Cloud Topic 3



- Word Cloud for Topic 4

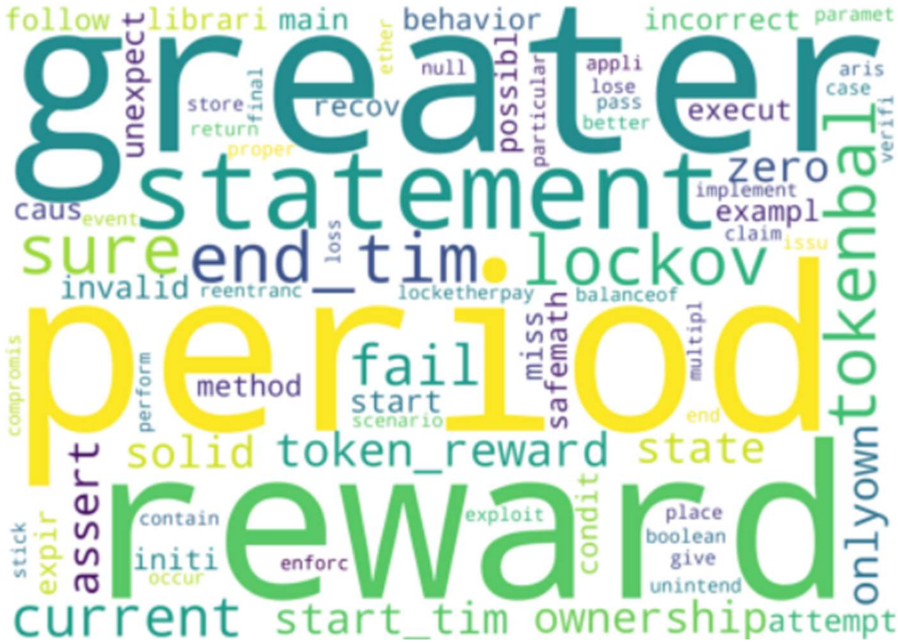
- Word Cloud for Topic 5

- Word Cloud for Topic 6



- Word cloud for Topic 7

- Word Cloud for Topic 8



- Word Cloud for Topic 9

5.3. Intertopic Distance Mapping

The intertopic distance map is a visualization technique used to understand the relationships between topics in a topic model. It is a scatter plot that shows the distance between topics in a two-dimensional space, based on the similarity of the words they contain.

the intertopic distance map is generated using the pyLDAvis library. This library provides a convenient way to visualize topic models, and is specifically designed to work with models trained using the Gensim library.

The first step in generating the intertopic distance map is to prepare the data for visualization using the pyLDAvis.gensim_models.prepare() function. This function takes three arguments:

lda_model: The trained LDA model.

bow_corpus: The bag-of-words corpus used to train the model.

dictionary: The dictionary used to create the bag-of-words corpus.

Once the data has been prepared, the pyLDAvis.enable_notebook() function is called to enable notebook output. This allows the visualization to be displayed directly in a Jupyter notebook.

Finally, the vis variable is assigned the result of calling pyLDAvis.gensim_models.prepare() with the appropriate arguments. This variable can then be displayed to generate the intertopic distance map.

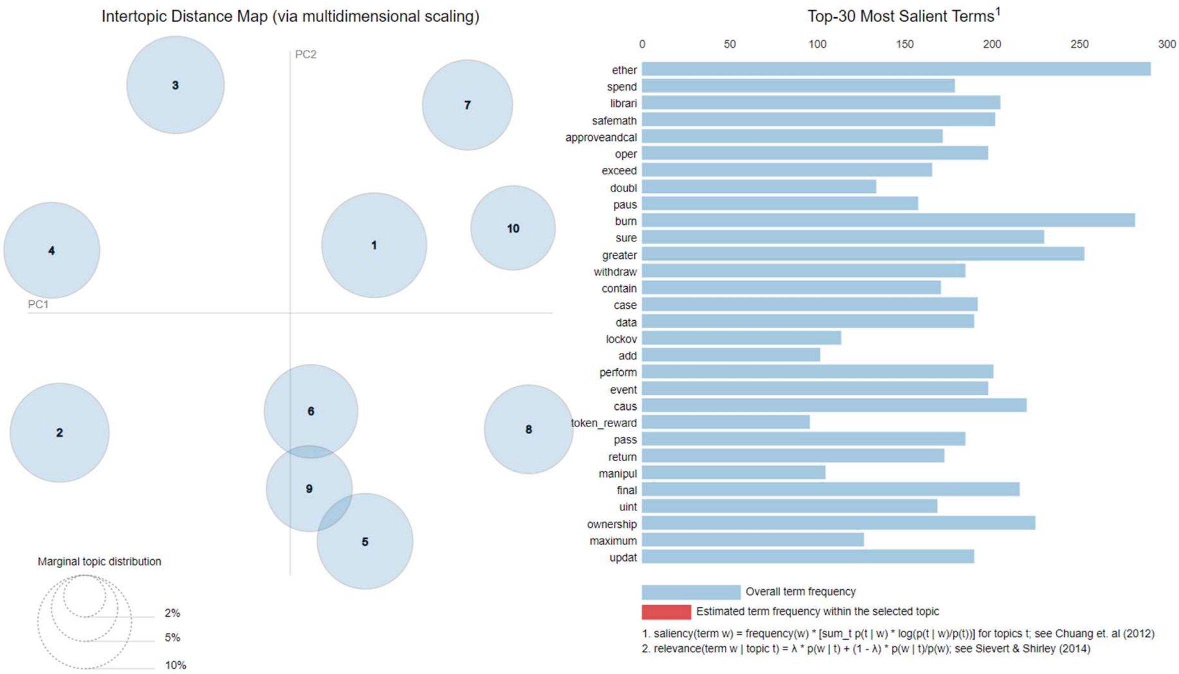


Figure 11 Intertopic Distance Mapping Visualization 1

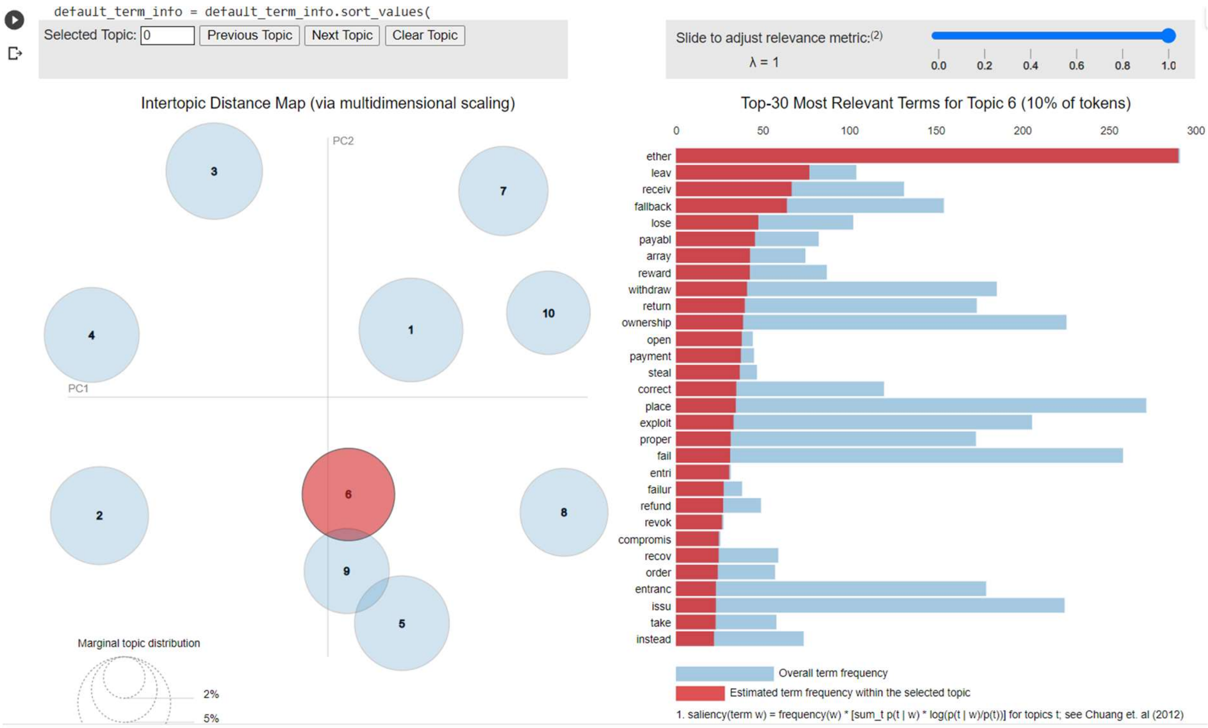


Figure 12 Intertopic Distance Mapping Visualization 2

Figures 11 and 12 are the Intertopic Distance Mapping Visualization which illustrates closeness of topics to each other and relationships between different topics that emerge from a corpus of text. The resulting map can be used to explore the relationships between topics and to identify clusters of related topics. It can also be used to identify gaps in the topics being discussed and to suggest new areas of research or inquiry.

Intertopic distance mapping is a powerful tool for visualizing and understanding complex relationships between topics in large collections of text.

In the pyLDAvis interface, there is a left panel that displays the topics in a model as circles. The size of each circle reflects the relative statistical importance of the topic. If you click on a circle or type in a topic number in the search field, you can select a particular topic to investigate. This left panel is also referred to as an "intertopic distance map (multidimensional scaling)." It helps users visualize how closely or distantly related the topics are to each other in statistical terms.

The right panel of the pyLDAvis interface displays the top words related to the selected topic on the left panel, accompanied by bar graphs that indicate their weight. The blue bar denotes the frequency of the word in the overall topic model, while the red bar signifies the frequency of the word within the selected topic. The right panel also features a relevance metric slider at the top, which arranges the words for a topic based on their importance.

One way to measure the association of a word with a topic is to consider its frequency in that specific topic. In other words, if the word appears more frequently in the topic, it is highly associated with that topic. The slider in pyLDAvis has a lambda (λ) value, which is initially set to "1." This setting arranges the words in the topic by their frequency, with longer red bars indicating higher frequency.

5.4. Results from Sentiment Analysis

The Table 2 shows the outcome form sentiment analysis the code prints the topics and their corresponding words and weights and writes them to a CSV file along with the sentiment scores. The table illustrates the negative, neutral and positive scores of the contracts. The compound score is calculated by normalizing the scores of the other three categories and then computing a sum using specific weights. The compound score is a weighted average of the normalized scores, where the weights are designed to place more emphasis on the most extreme scores. the SentimentIntensityAnalyzer module from NLTK is used to compute the polarity in the text which makes is easy to pick up which contracts seems more vulnerable.

Contract Number	Results from Sentiment Analysis			
	Negative	Neutral	Positive	Compound
0	0.049	0.883	0.067	0.25
1	0	0.934	0.066	0.4019
100	0.141	0.818	0.042	-0.6874
101	0.229	0.712	0.059	-0.875
102	0.172	0.714	0.114	-0.5423
104	0.139	0.789	0.072	-0.7212
107	0.184	0.695	0.121	-0.6033
108	0.04	0.96	0	-0.1531
113	0.024	0.935	0.041	0.1531
114	0.142	0.826	0.032	-0.8074
115	0.077	0.824	0.099	0.2658
117	0.106	0.872	0.022	-0.8122
118	0.087	0.722	0.191	0.7717
119	0.049	0.951	0	-0.3818
120	0.048	0.889	0.063	0.1842
121	0.062	0.889	0.049	-0.3523
122	0.075	0.859	0.066	-0.211
123	0.067	0.889	0.043	-0.4215
126	0.078	0.892	0.03	-0.4829

Table 2 Sentiment Analysis Results

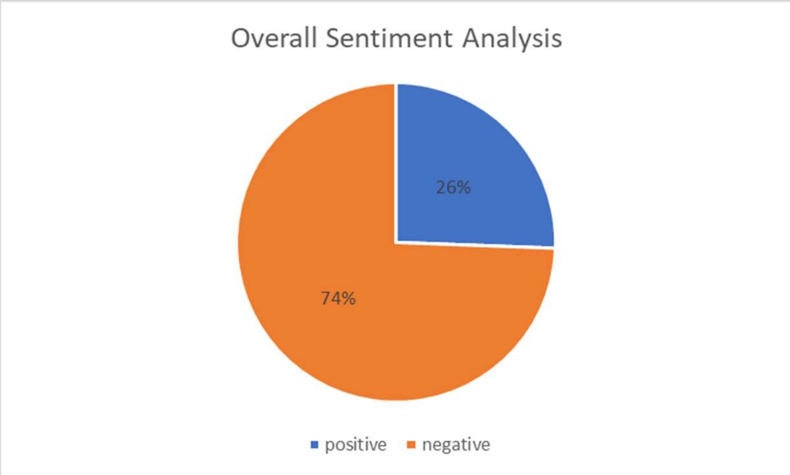


Figure 13 overall percentage of smartcontracts thar have positive and negative Overall Sentiments

From Figure 13 it can be inferred that 74% of smart contracts from our dataset that have overall sentiment as negative from sentiment analysis and are most likely to be vulnerable to attacks. The overall sentiments are the compound scores calculated as shown in the table, it is a single number between -1 and 1 that represents the overall sentiment of the text, with -1 indicating very negative sentiment and 1 indicating very positive sentiment.

5.5. Comparative Analysis with Slither

To check our results from ChatGPT and sentiment analysis, a comparative analysis was carried out on three smart contracts. Slither a very popular smart contract analysis tool was used to analyse these contracts and the results are compared as follows in tables 3,4 and 5.

Contract No.	**Chat GPT Results**	**Sentiment Analysis**	**Results from Slither**
102	Weaknesses: 1. The contract does not include any protection from replay attacks. 2. No authorization is checked before setting or getting data, anyone could change the reference or get the reference value. 3. No validation for the reference values are present. 4. There is no mechanism to safely	Negative: 0.172 Positive: 0.114 Neutral: 0.714 Compound: -0.542 Overall: Negative	1. high-confidence warning about a variable named "reference" in the SimpleAudit contract. Slither has detected that this variable name is already a built-in symbol in Solidity and advises the developer to avoid using such names to prevent potential naming conflicts. 2. high-confidence warnings about the SimpleAudit.set and SimpleAudit.get functions. Slither has identified that these functions should be declared as external functions, which can

	delete or update the data stored in the audit log. 5. There is no method to check the integrity of the stored data since there is no hash or signature verification.		improve the security and gas efficiency of the contract. 3. high-confidence warning about naming conventions. Slither has detected that the parameters "goeureka_audit_ref" in the SimpleAudit.set and SimpleAudit.get functions are not in mixed case, which can make the code less readable and harder to maintain.

Table 3 Comparative Analysis of Contract 102

Table 3- The analysis from this contract seems quite accurate from chatgpt as it aptly points out the issue of the integrity of the code and the way the code is written which is confirmed by Slither in last findings that the code is less readable and harder to maintain. Moreover the sentiment analysis gives overall rating as negative as the contract has high confidence warnings confirmed with Slither.

Contract No.	Chat GPT Results	Sentiment Analysis	Results from Slither
107	1. There is no restriction on how much ERC20 tokens can be minted, allowing for inflation. 2. No safety mechanism is built into the contract to check for overflow/underflow when subtracting or adding values. 3. No way to freeze accounts 4. There is no way to ensure funds are used properly as there is no audit trail of the transactions. 5. The transfer() and transferFrom() functions are not restricted to a certain amount of tokens, allowing for unlimited transfers.	Negative: 0.184 Positive: 0.121 Neutral: 0.695 Compound:-0.6 Overall: Negative	1. TokenERC20.decimals should be constant": this is an informational issue, indicating that the decimals variable in the TokenERC20 contract should be declared as a constant. 2. "TokenERC20.transfer should be declared external": this is an informational issue, indicating that the transfer function in the TokenERC20 contract should be declared as external. 3. "TokenERC20.approveAndCall should be declared external": this is an informational issue, indicating that the approveAndCall function in the TokenERC20 contract should be declared as external. 4. "TokenERC20.burn should be declared external": this is an informational issue, indicating that the burn function in the TokenERC20 contract should be declared as external. 5. "Function 'TokenERC20.TokenERC20' is not in mixedCase": this is an informational issue, indicating

			that the TokenERC20 constructor in the 107.sol file should follow the mixedCase naming convention.

Table 4 Comparative Analysis of Contract 107

Table 4-In this case the results from ChatGPT identifies more dangerous vulnerabilities like that there is no restriction on how many TokensERC20 can be minted that can lead to Reentrancy attacks and their vulnerabilities like Integer overflow/underflow. However, Slither addresses these issues technically.

Contract No.	**Chat GPT Results**	**Sentiment Analysis**	**Results from Slither**
2908	There are no weaknesses in the given code. However, there are some areas that could be improved upon in terms of readability and maintainability. For example, functions such as balanceOf, transferFrom, approve and allowance could be consolidated into a single function to reduce complexity, and comments can be added for better readability.	Negative: 0.081 Positive: 0.104 Neutral: 0.815 Compound: 0.378 Overall: Positive	The findings relate to several functions and properties within the contract, including totalSupply, balanceOf, allowance, transferFrom, approve, and transfer. The findings suggest that these functions should be declared as external and that certain properties, including Totalsupply and no_of_tokens, should be declared as constant.

Table 5 Comparative Analysis of Contract 2908

As for the case depicted in Table 5; a contract for which results from ChatGPT showed that the contract has no major vulnerabilities was selected and the sentiments were also positive, the results were accurately confirmed by Slither as it also did not show any major high confidence warnings in the smart contract.

Chapter 6

6. Conclusion

6.1. Future works and application

While this thesis has demonstrated the potential of using the OpenAI ChatGPT API for the analysis of Ethereum smart contracts, there are several avenues for future research that could improve the effectiveness and accuracy of the analysis.

Firstly, it would be useful to investigate the use of other NLP techniques, such as Named Entity Recognition (NER) and Coreference Resolution, to improve the accuracy of the topic and sentiment analysis. These techniques can help to identify specific entities mentioned in the contracts and resolve references to them, which can provide more detailed insights into the content of the contracts and the sentiments expressed towards them.

Secondly, it would be beneficial to explore the use of more advanced machine learning models, such as Transformer-based models like BERT or GPT-3, which have demonstrated state-of-the-art performance in NLP tasks. These models can provide more accurate and nuanced analysis of the text data and can potentially identify more subtle patterns and trends in the contracts [26].

Another potential area for future research is the development of a more comprehensive and diverse dataset of smart contracts for analysis. The current study used a limited dataset of Ethereum smart contracts, but expanding the dataset to include smart contracts from other blockchain platforms and industries can provide a more comprehensive understanding of the risks and vulnerabilities associated with smart contracts.

Additionally, it would be useful to investigate the potential of using the ChatGPT API for the analysis of other blockchain-related data, such as transaction data or network activity. This can provide insights into the behaviour and patterns of blockchain networks and can potentially identify anomalies or malicious activity.

Finally, it would be beneficial to develop a user-friendly interface or tool that can integrate the analysis capabilities of the ChatGPT API and provide a more accessible and streamlined approach to analysing smart contracts. This can help to promote the responsible use of smart contracts and increase transparency and trust in their use in various industries.

6.2. Final Word

In conclusion, while this thesis has demonstrated the potential of using the OpenAI ChatGPT API for the analysis of Ethereum smart contracts, there are several avenues for future research that can improve the accuracy and effectiveness of the analysis. These include the use of other NLP techniques, more advanced machine learning models, a more diverse dataset, the analysis of other blockchain-related data, and the development of a user-friendly interface or tool. Further research in these areas can help to promote the responsible use of smart contracts and increase transparency and trust in their use in various industries.

In conclusion, this thesis has demonstrated the effectiveness of the OpenAI ChatGPT API in analysing Ethereum smart contracts through the use of advanced NLP techniques, such as topic analysis and sentiment analysis. The findings of this research have potential implications for the field of blockchain and smart contract analysis, as it shows the feasibility of using automated NLP techniques to increase transparency and trust in the use of smart contracts in various industries.

The use of smart contracts has grown rapidly in recent years, offering numerous benefits to businesses across a range of industries. However, the potential for vulnerabilities and errors in these contracts means that there is a need for effective methods to identify and mitigate these risks. The approach presented in this thesis provides an automated and accessible solution to the problem, utilizing the power of NLP to extract relevant information and provide insights into the risks associated with smart contracts.

One of the key advantages of this technique is its ability to make smart contract analysis accessible to individuals who may not have prior knowledge or technical expertise in the field. By utilizing the ChatGPT API to perform topic and sentiment analysis, the risks associated with smart contracts can be easily understood and communicated to a wider audience. This has the potential to increase transparency and trust in the use of smart contracts, making them a more widely accepted solution for businesses across various industries.

Furthermore, the automation of the analysis process using the ChatGPT API significantly reduces the time and effort required for manual review, allowing for a more comprehensive analysis of smart contracts. This approach has the potential to save businesses time and resources in the long term, while also improving the overall security and reliability of smart contracts.

The findings of this thesis also demonstrate the effectiveness of using advanced NLP techniques to analyze and understand the risks associated with smart contracts. Through the use of topic analysis, the main themes discussed in the contracts can be identified, while sentiment analysis provides insight into the emotions and opinions expressed in the text. This information can be used to identify potential risks and vulnerabilities in the contracts, such as security or privacy issues, and to make informed decisions regarding their use.

The potential implications of this research extend beyond the field of blockchain and smart contract analysis, with the technique presented having the potential to be applied to a range of text-based data analysis problems. The success of this approach highlights the potential of advanced NLP techniques to offer valuable insights into large volumes of text data, opening up new possibilities for automated analysis and decision-making in a range of industries.

In summary, this thesis has shown the potential of using the OpenAI ChatGPT API to analyze Ethereum smart contracts through the use of advanced NLP techniques. By providing an accessible and automated solution to the problem of smart contract analysis, this approach has the potential to increase transparency and trust in the use of smart contracts in various industries. The findings of this research demonstrate the capabilities of advanced NLP techniques to provide valuable insights into large volumes of text data, opening up new possibilities for automated analysis and decision-making in a range of fields. As the use of smart contracts continues to grow, the importance of effective risk analysis and mitigation strategies will only increase, making the findings of this research even more relevant and impactful.

Bibliography

[1] Bonneau, Felten, Narayanan, “Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction. Princeton University Press.”.

[2] Tschorsch, Scheuermann, Bitcoin and Beyond: A Technical Survey on, IEEE COMMUNICATIONS SURVEYS & TUTORIALS, VOL. 18, NO. 3, THIRD QUARTER 2016.

[3] Maher, Kumar “Blockchain and the Transformation of the Supply Chain : An integrated theoretical perspective of organizational adoption.,” no. Journal of Business Research, 98, 365-380..

[4] Kumar, Hasan, Rashid “Blockchain in Healthcare: A Review, MDPI -”.

[5] M.Swan, “Blockchain: Blueprint for a New Economy. O'Reilly Media, Inc.”.

[6] Christin, Edelman, Böhme, “Bitcoin: Economics, Technology, and Governance. Journal of Economic Perspectives, 29(2), 213-238.,” (2015).

[7] M. Pilkington, “Blockchain Technology: Principles and Applications. Research Handbook on Digital Transformations, Edward Elgar Publishing, pp. 225-253.,” (2015).

[8] Verma, Pattanayak, Crosby, “Blockchain technology: Beyond bitcoin. Applied Innovation, 2(6-10), 71-81.,” (2016).

[9] V. Buterin, “thereum: A Secure Decentralized Generalized Transaction Ledger. Ethereum Foundation.,” (2014)..

[10] Ethereum., “Ethereum Virtual Machine.,” (2022). . [Online]. Available: https://ethereum.org/en/developers/docs/evm/.

[11] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper, 151, 1-32.,” 2014.

[12] “Solidity documentation:,” [Online]. Available: https://docs.soliditylang.org/en/v0.8.11/.

[13] R. Modi, “Solidity Programming Essentials: A beginner's guide to smart contract development on the Ethereum blockchain,” Packt Publishing, 2018.

[14] [Online]. Available: https://www.naukri.com/learning/articles/solidity-programming-language/.

[15] “SmartBugs A Framework to Analyze Solidity Smart Contracts for Security Vulnerabilities.,” 2018. [Online]. Available: : https://arxiv.org/abs/1805.07501.

[16] Atzei, Bartolleti, Cimoli “ A survey of attacks on Ethereum smart contracts (No. arXiv:1710.06169),” no. Cornell University, 2017.

[17] Makhija, "Ethereum smart contract security using natural language processing. In 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB) (pp. 22-29). IEEE.," 2019.

[18] Ramedar, Angelo, Salzer "Review of Automated Vulnerability Analysis of Smart Contracts on Ethereum," 2020.

[19] Zheng, Xie, Ning Dai, Chen "A review on blockchain-based smart contracts: Challenges and opportunities. Future Generation Computer Systems, 102, 475-491.," 2020.

[20] Arif Furkan Mendi, "A Sentiment Analysis Method Based on a Blockchain-Supported Long Short-Term Memory Deep Network. MDPI.," 2020.

[21] Alzubi, Amed, Kahtan, Ashraf "Internet of Things and Blockchain Integration: Security, Privacy, Technical, and Design Challenges" 2021.

[22] A. S. N. Lodha, "Investigating the Role of Artificial Intelligence in," no. 2022 International Conference on Emerging Smart Computing and Informatics (ESCI).

[23] RAJESH GUPTA 1, "Smart Contract Privacy Protection Using AI in Cyber-Physical Systems: Tools,Techniques and Challenges".

[24] Zha, Li, Liu, Meng, Fang, "Countermeasure Based on Smart Contracts and AI against DoS/DDoS Attack in 5G circumstances".

[25] "Smart-Contract-Dataset," [Online]. Available: https://github.com/Messi-Q/Smart-Contract-Dataset#download-dataset.

[26] https://www.coding-bootcamps.com/blog/how-ethereum-virtual-machine-works.html.

[27] D.Caldarini,"Topic Modelling con Gensim," [Online]. Available: https://techblog.smc.it/it/2021-10-22/topic-modelling.