AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY

FACULTY OF ELECTRICAL ENGINEERING, AUTOMATICS, COMPUTER SCIENCE AND BIOMEDICAL ENGINEERING

Master Thesis

Machine Learning Methods in Smart Contract Security

Author: *Vyacheslav Trushkov*
Degree programme: *Computer Science and Intelligent Systems*
Supervisor: *Sebastian Ernst PhD Eng.*

Krakow, 2023

Thank you very much to Sebastian Ernst PhD Eng., Konrad Zaworski, MSc Eng., parents, family, and friends for their boundless support and shared knowledge.

Contents

1. Introduction

Blockchain technology represents a groundbreaking innovation in the world of digital transactions. It operates as a distributed ledger system, recording transactions securely and immutably across a network of computers. This stands in stark contrast to traditional centralized systems, where a single entity controls the ledger. Blockchains, on the other hand, embrace decentralization, transparency, and tamper resistance as their core principles.

At the heart of blockchain technology lies the concept of a "block" – a container for a set of transactions. These blocks are cleverly linked together using cryptographic hashes, forming what we commonly refer to as a "blockchain". This architecture, with its decentralized and transparent characteristics, makes blockchain an ideal platform for a wide range of applications, including cryptocurrencies, supply chain management, and, most notably for our discussion, smart contracts.

Smart contracts represent the pinnacle of this blockchain revolution. These self-executing agreements, encoded in lines of code, possess the remarkable ability to automatically execute transactions once predefined conditions are met. This groundbreaking feature eliminates the need for intermediaries, streamlines processes, and substantially reduces the potential for human errors and fraudulent activities. The Ethereum platform has emerged as a prominent ecosystem for the development and deployment of these smart contracts, thanks to its Turing-complete[1] *Ethereum Virtual Machine* (EVM)[2] and support for custom programming languages such as *Solidity*[3].

However, the increasing ubiquity of smart contracts has brought with it a host of security concerns. Given the immutable nature of smart contracts and the often substantial financial assets they manage, any vulnerability or flaw in the contract's code can lead to severe financial losses or other detrimental consequences. This vulnerability has been underscored by several high-profile incidents, including the DAO hack in 2016 and the Parity wallet vulnerabilities in 2017, which resulted in millions of dollars in losses.

[1] In computability theory, a system of data manipulation rules is considered Turing-complete or computationally universal if it has the capability to emulate any Turing machine.

[2] `https://ethereum.org/en/developers/docs/evm/` (visited on 2023-04-06)

[3] `https://soliditylang.org` (visited on 2023-04-06)

Given these challenges, the quest for robust smart contract security has become a pressing concern in the blockchain community. A variety of methods have been proposed to tackle this issue, ranging from formal verification and static analysis to runtime monitoring. Recently, Machine Learning (ML) techniques, particularly those based on transformer models, have shown considerable promise in the analysis and detection of smart contract vulnerabilities. The transformer architecture has revolutionized the field of *Natural Language Processing* (NLP) and has demonstrated remarkable performance across a wide range of tasks, including machine translation, sentiment analysis, and more [4].

Machine learning (ML) is a subfield of artificial intelligence (AI) that uses algorithms and statistical models to enable computers to learn from data and make predictions or decisions without being explicitly programmed. ML has been successfully applied to a wide range of tasks, including natural language processing, image recognition, and, more recently, the blockchain security.

The decision to focus specifically on Ethereum and the mentioned ML methods is driven by a twofold rationale. Firstly, Ethereum has witnessed the majority of significant smart contract-related incidents and security breaches to date. This concentration of real-world vulnerabilities and attacks on the Ethereum platform provides a rich and pertinent dataset for research. Moreover, Ethereum's popularity has led to the creation of numerous smart contracts on its platform within the broader EVM-based blockchain ecosystem, making the findings and solutions derived from Ethereum research highly transferable to other EVM-based blockchains.

Secondly, the choice of ML methods, particularly transformer models, is motivated by their exceptional performance in processing and understanding complex patterns within data. These models have showcased their effectiveness in tasks that involve analyzing and interpreting the intricate semantics of code, making them a natural fit for the analysis of malicious smart contracts.

1.1. Objective and scope of the thesis

The objective of this thesis is to develop a model for the detection of malicious smart contracts using state-of-the-art machine learning techniques. The primary focus is to create an effective model that can identify and classify potentially harmful or malicious code.

The thesis will primarily focus on the Ethereum platform and the detection of malicious smart contracts developed in Solidity. It will explore the application of machine learning models, with a particular emphasis on transformer models, such as the *Bidirectional Encoder Representations from Transformers* (BERT). The study will investigate the technical intricacies of transformer models and their suitability for analyzing smart contract code.

The research will involve tasks such as reviewing the current state of knowledge in the field of smart contract security, collecting relevant data for model training, implementing and fine-tuning the machine learning model, and evaluating its performance. The model's effectiveness will be assessed based on its ability to accurately detect malicious code.

1.2. Thesis structure

The thesis is structured as follows:

- **Chapter 2** provides a background on the Ethereum Virtual Machine (EVM), smart contracts, and the DeFi ecosystem. It also discusses the escalation of smart contract exploits and the role of machine learning in blockchain security.
- **Chapter 3** delves into the transformer-based machine learning models for smart contract analysis, with a focus on the BERT model.
- **Chapter 4** details the technical implementation of the model, including data collection and preprocessing, model training and evaluation, and model deployment and usage.
- **Chapter 5** presents a real-world attack detected by the bot, discussing the attack on Hundred Finance, its aftermath, and the lessons learned.
- **Chapter 6** concludes the thesis with a summary of the objectives achieved, potential for further development, and a discussion on the results and limitations. It also highlights the need for additional indicators to distinguish real threats from noise in the blockchain ecosystem and the importance of a response strategy.

2. Background and theoretical aspects

2.1. Ethereum Virtual Machine (EVM)

2.1.1. Overview

The Ethereum Virtual Machine (EVM) plays a vital role in the Ethereum blockchain as a decentralized, Turing-complete virtual machine. It acts as the computational engine that powers the execution of smart contracts, which are self-executing agreements encoded directly into lines of code. By processing, verifying, and enforcing the rules established by these contracts, the EVM ensures the security and determinism of transactions and state changes on the Ethereum network.

As a decentralized virtual machine, the EVM operates across a network of computers, known as nodes, spread throughout the world. Each node independently executes the instructions within smart contracts, ensuring consensus and agreement on the outcome of the computation. This distributed nature of the EVM contributes to the resilience and immutability of the Ethereum blockchain, as no single entity has control over the execution or manipulation of smart contract code.

The EVM's Turing completeness allows for the execution of complex computational tasks within smart contracts, making it capable of handling a wide range of applications. From decentralized finance protocols to non-fungible token marketplaces and decentralized applications, the EVM enables the development and execution of diverse functionalities on the Ethereum blockchain.

By processing transactions and executing smart contracts, the EVM maintains the integrity and security of the Ethereum network, ensuring that all participants have trust in the outcomes and enforcing the predefined rules of the blockchain. Its deterministic nature guarantees that given the same inputs, the EVM will always produce the same results, promoting consistency and predictability in the execution of smart contract logic[1].

[1] `https://ethereum.org/developers/docs/evm` (visited on 2023-04-07)

2.1.2. Functioning and execution

The EVM operates on a stack-based architecture, utilizing a simple set of operations, such as arithmetic, logical, and control flow instructions, to execute smart contract code. The code is written in a low-level, bytecode format known as Ethereum bytecode, which is the compiled version of higher-level programming languages like Solidity or *Vyper* [2].

A fundamental concept in the EVM is gas, which serves as the measure of computational effort required to perform various operations. Each operation within the EVM consumes a specific amount of gas, and this consumption is used to meter and limit the execution of smart contracts [2]. We will analyze this concept in the next subsection.

The EVM operates on an account-based model, comprising *Externally Owned Accounts* (EOAs) and contract accounts. EOAs, controlled by private keys, are capable of initiating transactions, while contract accounts store smart contract code and associated data. The EVM maintains a global state that maps account addresses to their respective account states, encompassing the account's nonce, balance, code, and storage [3]. For a visual representation of this model, please refer to figure 2.1.

[2] `https://www.lcx.com/ethereum-virtual-machine-evm/` (visited on 2023-04-09)

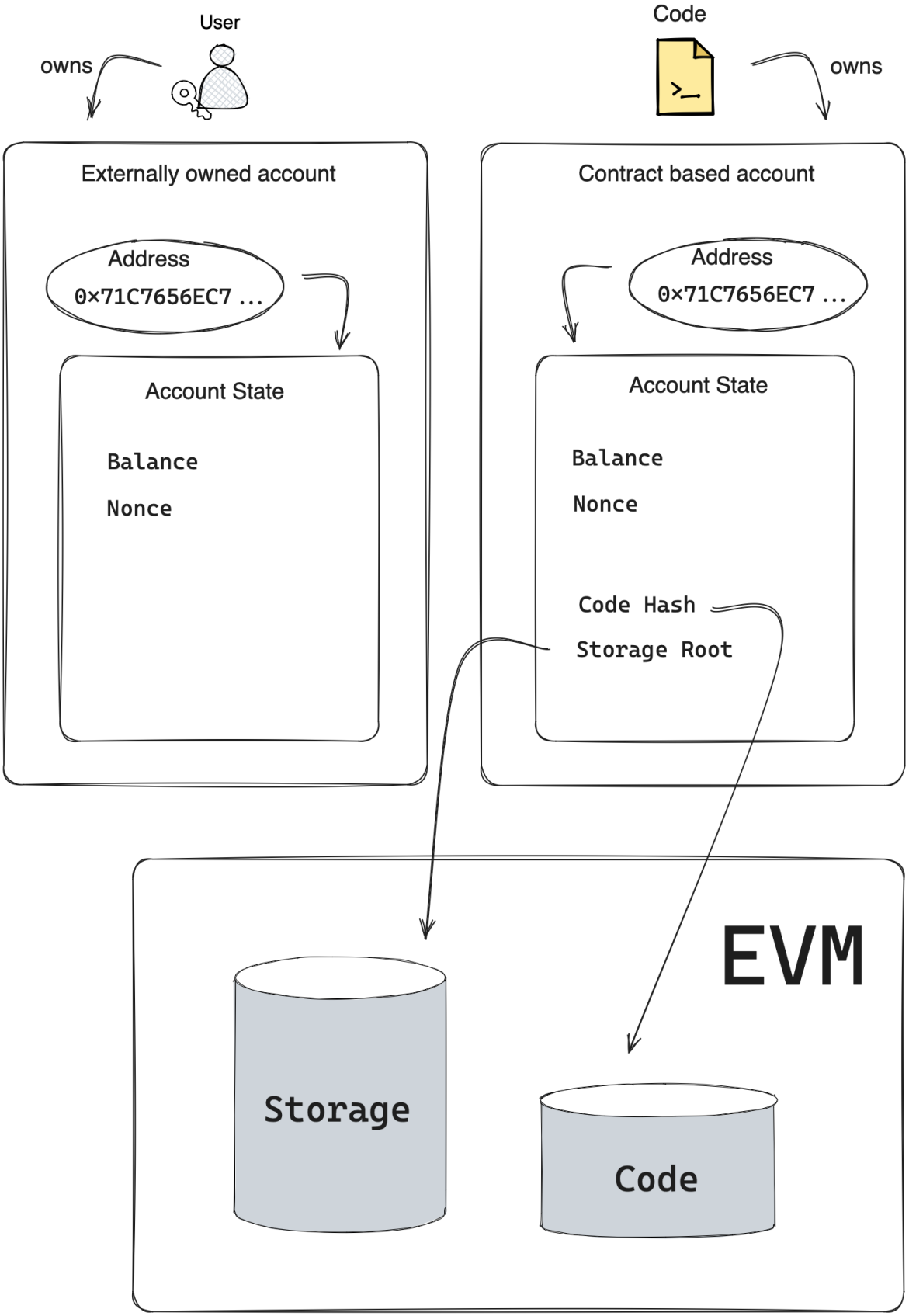

Fig. 2.1. Ethereum account types EOA and CA

The Ethereum Virtual Machine (EVM) plays a pivotal role in the Ethereum ecosystem by providing a secure and deterministic environment for the execution of smart contracts. Its stack-based architecture, gas system, and account-based model facilitate the development and deployment of decentralized applications and pave the way for a wide range of use cases within the blockchain space [3].

2.1.3. Gas in Ethereum Virtual Machine (EVM)

The concept of gas in Ethereum can be understood through two key components:

1. **Gas Cost** – This is a constant value for an operation or computation, determined by the complexity of the operation. For example, a simple operation like addition or subtraction has a lower gas cost than a more complex operation like multiplication or exponentiation.

2. **Gas Price** – this is the amount of ETH[4] a user is willing to spend per unit of gas. It is usually measured in Gwei, where $1 \text{ Gwei } = 10^{-9}$ ETH. The gas price is not constant and can fluctuate based on network demand. When the network is busy, users often increase the gas price of their transactions to incentivize validators to prioritize their transactions.

The total transaction fee (in Ether) that a user needs to pay for a transaction or smart contract execution is calculated as the product of the gas cost and the gas price:

$$\text{Transaction Fee } = \text{ Gas Cost } \times \text{ Gas Price}$$

Users initiating transactions or invoking smart contract functions are required to specify a gas limit and a gas price. The gas limit is the maximum amount of gas the user is willing to consume for the transaction. If the gas limit is exceeded during the execution, the transaction or contract call will revert, and the consumed gas will be forfeited as a fee to the miners.

It's important to note that unused gas from the gas limit is refunded to the user, so it's generally safe to set a higher gas limit. However, setting a high gas price can result in higher transaction fees. The concept of gas in the EVM is crucial for managing and limiting computational work, preventing spam on the network, and incentivizing validators to validate and include transactions in the blockchain[5].

[3] `https://101blockchains.com/ethereum-virtual-machine` (visited on 2023-04-20)
`https://medium.com/@blocktorch/architecture-of-decentralized-applications-dapps-d583db198a6f` (visited on 2023-04-20)

[4] "ETH" refers to the native cryptocurrency of the Ethereum blockchain. It is commonly referred to as "Ether" and similar to how Bitcoin (BTC) is the digital currency associated with the Bitcoin blockchain, ETH is specifically designed for use within the Ethereum ecosystem. As of June 26, 2023, the price of 1 ETH is approximately $1,850 USD.

[5] `https://ethereum.org/en/developers/docs/gas/` (visited on 2023-04-23)

During normal network conditions, gas prices can range from 10 to 50 Gwei. However, during periods of high network congestion or increased demand, gas prices can spike significantly, occasionally reaching levels as high as 400-500 Gwei or even more.

Users initiating a simple send transaction of Ether (ETH) typically require a gas limit of around 21,000 units of gas. This value represents the base cost for a basic transaction on the Ethereum network. The gas cost for a simple send transaction can range from approximately 0.00042 ETH to 0.0021 ETH, depending on the prevailing gas price.

Creating a contract on the Ethereum network involves a more complex operation and requires additional computational resources. The gas cost for contract creation can vary based on the complexity of the contract's code and the number of operations involved. On average, contract creation can range from approximately 300,000 to 2,000,000 gas units, translating to a gas cost of around 0.015 ETH to 0.1 ETH.

Sending a Non-Fungible Token (NFT) or performing a swap on Decentralized EXchanges (DEX) typically involves interacting with smart contracts and executing more complex transactions. Gas costs for these operations can vary depending on the specific contract and the complexity of the transaction. On average, the gas cost for sending an NFT or performing a swap can range from approximately 100,000 to 500,000 gas units, equivalent to a gas cost of around 0.005 ETH to 0.025 ETH.

2.2. Smart contracts

2.2.1. Definition and characteristics

Smart contracts are self-executing contracts with the terms of the agreement directly encoded into computer code. They are stored and executed on a blockchain, ensuring immutability, transparency, and decentralization. Smart contracts automatically execute the specified actions when the predefined conditions are met, thus eliminating the need for intermediaries and reducing the potential for human error or fraud [9].

The concept of smart contracts was first proposed by Nick Szabo in 1994 [1], but it wasn't until the advent of the Ethereum platform that the concept gained significant attention and traction. Ethereum's Turing-complete EVM and support for custom programming languages like Solidity have facilitated the development and deployment of a wide range of smart contract use cases, from decentralized finance (DeFi) to supply chain management and digital identity [13].

Key characteristics of smart contracts include:

- **Immutability** – Once a smart contract is deployed on the blockchain, it cannot be altered or tampered with. This ensures the integrity and trustworthiness of the contract's terms and conditions.
- **Transparency** – All transactions and state changes related to a smart contract are publicly visible on the blockchain, providing a transparent record of the contract's execution.
- **Decentralization** – Smart contracts are executed by multiple nodes in the blockchain network, ensuring that the contract's outcome is agreed upon through a decentralized consensus mechanism.
- **Automation** – Smart contracts automatically execute the specified actions when the predefined conditions are met, reducing the need for manual intervention and enabling more efficient processes.

2.2.2. Advantages and challenges

Smart contracts offer several advantages over traditional contracts, including:

- **Cost-efficiency** – By automating processes and eliminating intermediaries, smart contracts can reduce transaction costs and improve overall efficiency.
- **Speed** – The automated execution of smart contracts can significantly reduce the time required for contract fulfillment and dispute resolution.
- **Security** – The decentralized nature and cryptographic security of blockchain technology can help minimize the risk of fraud, hacking, or other malicious activities.
- **Trust** – The transparency and immutability of smart contracts can help build trust among parties, as the terms of the contract are publicly visible and cannot be altered after deployment.

However, smart contracts also face several challenges, such as:

- **Scalability** – As the number of smart contracts and transactions on a blockchain network grows, the network may struggle to handle the increased load, leading to slower transaction times and higher fees.
- **Privacy** – The transparency of smart contract transactions may raise privacy concerns for certain use cases or industries.

– **Legal and regulatory issues** – The legal status and enforceability of smart contracts vary across jurisdictions, and regulatory frameworks for blockchain technology are still evolving.

– **Security vulnerabilities** – Despite the security benefits of smart contracts, poorly written or flawed code can expose the contract to vulnerabilities, as demonstrated by high-profile hacks and incidents in the past.

It's worth noting that as smart contracts are executed and added to the blockchain, the size of the blockchain increases over time. This growth in size has implications for the storage and maintenance of the blockchain data. While the Bitcoin blockchain has a fixed block size, the Ethereum blockchain has faced challenges in managing the increasing size due to the inclusion of smart contracts and their associated data. This issue has prompted discussions and proposals for various scalability solutions, such as Ethereum 2.0 and layer 2 scaling solutions, to address the growing storage requirements while maintaining network performance.

As we can see smart contracts offer numerous advantages over traditional contracts in terms of cost, speed, security, and trust. However, challenges related to scalability, privacy, legal and regulatory issues, and security vulnerabilities must be addressed to ensure the widespread adoption and success of smart contracts in various industries and use cases [8].

2.3. Decentralized Finance (DeFi)

2.3.1. Concept and significance

Decentralized Finance, commonly known as DeFi, is a financial ecosystem built on blockchain technology using smart contracts. It aims to bypass traditional financial intermediaries like banks and brokerages, creating a more accessible, transparent, and efficient financial system [10]. DeFi leverages the power of smart contracts, decentralized applications (DApps), and various blockchain protocols to facilitate a wide range of financial services, including lending, borrowing, trading, asset management, and insurance.

The significance of DeFi lies in its potential to democratize access to financial services, reduce costs, and enhance the efficiency of the global financial system. Some of the key benefits of DeFi include:

– **Permissionless access** – DeFi platforms typically do not require users to provide identification or undergo stringent KYC (Know Your Customer) checks. This allows people from around the world, including those without access to traditional banking services, to participate in the financial ecosystem.

– **Transparency** – Transactions and smart contract executions on DeFi platforms are recorded on a public blockchain, providing a transparent and auditable record of financial activities.

– **Interoperability** – DeFi protocols and DApps are often built on open-source frameworks and designed to be compatible with one another. This enables seamless integration and interaction between various platforms and services.

– **Composability** – The modular nature of DeFi allows developers to build complex financial products and services by combining existing protocols and DApps. This fosters innovation and rapid development within the ecosystem.

2.3.2. Applications and risks

DeFi encompasses a wide range of financial applications and services, such as:

– **Lending and borrowing platforms** like Aave and Compound enable users to lend or borrow cryptocurrencies, often with variable interest rates determined by supply and demand dynamics.

– **Decentralized EXchanges (DEXs)** like Uniswap and SushiSwap facilitate peer-to-peer trading of cryptocurrencies and other digital assets without the need for a centralized intermediary. They use automated market makers (AMMs) or order book-based systems to manage trades.

– **Stablecoins** like DAI and USDC are cryptocurrencies pegged to a stable asset, such as a fiat currency. They aim to minimize price volatility and provide a stable means of exchange and store of value within the crypto ecosystem.

– **DeFi platforms** like Yearn Finance and Balancer enable users to optimize their investment strategies, automate asset management, and participate in yield farming or liquidity mining opportunities.

Despite its numerous advantages, DeFi also faces several risks and challenges, including:

– **Security vulnerabilities** – DeFi platforms and smart contracts can be vulnerable to hacks, exploits, and other security threats, leading to significant financial losses.

– **Regulatory uncertainty** – The rapidly evolving nature of DeFi has raised concerns among regulators, and the legal status and compliance requirements for DeFi platforms and services remain unclear in many jurisdictions.

– **Market volatility** – The DeFi ecosystem is often subject to high levels of price volatility, which can impact the stability of collateralized loans, liquidity pools, and other financial products.

– **Usability and complexity** – The complexity of DeFi platforms and services can make them difficult for non-expert users to navigate, potentially limiting widespread adoption.

Decentralized Finance (DeFi) represents a transformative shift in the financial industry, offering a more accessible, transparent, and efficient alternative to traditional financial services. However, to fully realize its potential, DeFi must address the challenges and risks associated with security vulnerabilities, regulatory uncertainty, market volatility, and usability. One of these risks leads us to the following section.

2.4. Smart contract code vulnerabilities and attacks

2.4.1. Vulnerabilities

Smart contract code vulnerabilities refer to weaknesses or flaws in the code that can be exploited by malicious actors to compromise the intended functionality or security of the contract. These vulnerabilities can arise due to programming errors, improper design, or inadequate security considerations. The vulnerabilities can be classified into different categories based on the types of risks they pose. Let's explore three common categories:

– **Reentrancy Attacks** is considered to be one of the most severe vulnerabilities for a smart contract. It occurs when a contract calls another contract and the called contract can invoke a function that reenters the calling contract before the initial call completes. This can lead to unexpected behavior and potential manipulation of the contract's state. Notably, the infamous "The DAO" attack in 2016 was a reentrancy attack. A simple example of a contract susceptible to reentrancy is the following:

```
contract EtherStore {
mapping(address => uint256) public balances;
bool private lockBalances;

    function deposit() public payable {
        require(!lockBalances);
        lockBalances = true;
        balances[msg.sender] += msg.value;
        lockBalances = false;
```

```
    }

    function withdraw(uint256 amount) public {
        require(!lockBalances && amount > 0 &&
        balances[msg.sender] >= amount);
        lockBalances = true;

        (bool success, ) = msg.sender.call{value: amount}("");

        if (success) {
            balances[msg.sender] -= amount;
        }

        lockBalances = false;
        return true;
    }
}
```

In this example, the `EtherStore` contract allows users to deposit and withdraw funds. However, the `withdraw` function is susceptible to reentrancy attacks. An attacker can create a malicious contract that repeatedly calls the `withdraw` function, draining the contract's balance before the deduction of the user's funds. To mitigate reentrancy attacks, developers should employ a mutex or reentrancy guard to prevent multiple reentrant calls. [**HarryErmawan**]

– **Integer Overflow/Underflow** vulnerabilities occur when arithmetic operations on integers result in values that exceed the maximum or minimum limits of the data type. This can lead to unexpected behavior or even bypassing certain checks. Consider the following vulnerable contract where the `transfer` function is vulnerable to integer overflow:

```
contract Token {
    mapping(address => uint256) public balances;

    function transfer(address to, uint256 value) public {
        require(balances[msg.sender] >= value &&
        balances[to] + value >= balances[to]);

        balances[msg.sender] -= value;
```

```
            balances[to] += value;
        }
    }
```

In this example, the `transfer` function allows users to transfer tokens to another address. However, it does not adequately check for potential integer overflow when updating the balance of the receiver. An attacker could craft a transaction that results in an integer overflow, allowing them to manipulate the contract's state. To address this vulnerability, developers should perform careful arithmetic checks to prevent overflow and underflow[6].

– **Unchecked external calls** refer to situations where a smart contract interacts with external contracts or addresses without properly validating or handling the returned data. This can allow attackers to manipulate the contract's execution flow or trick it into unintended actions. Consider the following vulnerable contract:

```
contract Lotto {
    bool public payedOut = false;
    address public winner;
    uint public winAmount;
    function sendToWinner() public {
        require(!payedOut);

        // Insecure external call
        winner.send(winAmount);

        payedOut = true;
    }

    function withdrawLeftOver() public {
        require(payedOut);

        // Insecure external call
        msg.sender.send(address(this).balance);
    }
}
```

[6] https://medium.com/hryer-dev/vulnerabilities-attacks-of-smart-contracts-9f112ea6c52c (vistied on 2023-04-25)

In this example, the `Lotto` contract has a vulnerability where the `sendToWinner` and `withdrawLeftOver` functions use the `send` method without properly checking the response. If a winner's transaction fails, it allows the `payedOut` flag to be set to **`true`**, regardless of whether Ether was sent or not. In this case, anyone can call the `withdrawLeftOver` function and withdraw the winner's winnings. To mitigate this vulnerability, developers should carefully validate the return value of external calls and handle any potential exceptions or failures appropriately[7].

2.4.2. Mechanisms of Attacks

To exploit smart contract vulnerabilities, attackers can carefully construct malicious contracts or inputs that trigger the vulnerable functions in unexpected ways. They can abuse reentrancy vulnerabilities, manipulate integer calculations, or exploit unchecked external calls to their advantage. Attackers analyze the target contract, identify its weaknesses, and design malicious code to trigger those vulnerabilities. They may aim to steal funds, manipulate contract states, or disrupt the intended functionality.

Let's consider the vulnerable contract called *EtherStore.sol*. This contract serves as an Ethereum vault, allowing depositors to withdraw only 1 ether per week. Here is the code:

```
contract EtherStore {
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() external payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds(uint256 _weiToWithdraw) public {
        require(balances[msg.sender] >= _weiToWithdraw);
        require(_weiToWithdraw <= withdrawalLimit);
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        require(msg.sender.call.value(_weiToWithdraw)());
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
    }
```

[7] `https://dev.to/kamilpolak/hack-solidity-unchecked-call-return-value-7og` (visited on 2023-05-01)

```
}
```

The vulnerability in this contract lies in `require(msg.sender.call.value(_weiToWithdraw)());` where the contract sends the requested amount of ether back to the user. An attacker can exploit this vulnerability by carefully constructing a malicious contract with a fallback function containing malicious code[8].

Let's examine the *Attack.sol* contract, which demonstrates the exploit:

```
import "EtherStore.sol";

contract Attack {
    EtherStore public etherStore;

    constructor(address _etherStoreAddress) {
        etherStore = EtherStore(_etherStoreAddress);
    }

    function attackEtherStore() external payable {
        require(msg.value >= 1 ether);
        etherStore.depositFunds.value(1 ether)();
        etherStore.withdrawFunds(1 ether);
    }

    function collectEther() public {
        msg.sender.transfer(address(this).balance);
    }

    fallback() external payable {
        if (etherStore.balance > 1 ether) {
            etherStore.withdrawFunds(1 ether);
        }
    }
}
```

[8] https://betterprogramming.pub/preventing-smart-contract-attacks-on-ethereum-a-code-analysis-bf95519b403a (visited on 2023-05-01)

In this example, the Attack contract is specifically designed to exploit the reentrancy vulnerability in the EtherStore contract.

To explain how the exploit works, let's walk through the steps:

1. The attacker deploys the malicious contract (Attack.sol) and provides the address of the EtherStore contract to be attacked as a constructor parameter.

2. The attacker calls the `attackEtherStore` function, sending at least 1 ether. This function calls the `depositFunds` function of the EtherStore contract to deposit 1 ether and then immediately triggers the `withdrawFunds` function with an amount of 1 ether.

3. As the `withdrawFunds` function is executed, it passes all the requirements, including the check for the time elapsed since the last withdrawal. The contract sends 1 ether back to the attacker's contract.

4. Upon receiving the ether, the fallback function of the attacker's contract is invoked. If the balance of the EtherStore contract is still greater than 1 ether, the fallback function calls the `withdrawFunds` function again, reentering the EtherStore contract.

5. The reentrancy attack continues as long as the balance of the EtherStore contract remains above 1 ether. The attacker can repeatedly withdraw funds until the balance is depleted.

To mitigate reentrancy vulnerabilities, developers should follow best practices such as:

– Using the transfer function instead of `call.value` when sending ether to external contracts to limit the gas stipend.

– Ensuring that all state changes occur before any ether transfers or external calls.

– Implementing the "checks-effects-interactions" pattern to separate checks, state changes, and interactions with external contracts.

– Introducing a mutex or reentrancy guard to prevent multiple reentrant calls.

By applying these techniques, developers can minimize the risk of reentrancy attacks and enhance the security of their smart contracts[9].

[9] `https://betterprogramming.pub/preventing-smart-contract-attacks-on-ethereum-a-code-analysis-bf95519b403a` (visited on 2023-05-01)

2.5. The escalation of smart contract exploits

The rapid growth and adoption of blockchain technology and cryptocurrencies have brought about a new era of financial innovation. However, this has also opened up opportunities for malicious actors to exploit vulnerabilities in smart contracts, leading to significant financial losses.

2.5.1. The rising tide of smart contract exploits

The frequency and severity of smart contract exploits have been on a steady rise over the past few years. According to preliminary statistics, monetary losses due to smart contract exploits in 2020 amounted to $215 million. This figure escalated dramatically in the following years, reaching $1.3 billion in 2021 and $2.7 billion in 2022 [19].

2.5.2. Yearly breakdown of financial losses

The year 2020 marked the beginning of a significant uptick in smart contract exploits. With a total of $215 million in losses, it was a wake-up call for the blockchain community about the potential risks associated with smart contracts.

In 2021, the situation worsened as the total losses due to smart contract exploits skyrocketed to $1.3 billion. This was a staggering six-fold increase from the previous year, highlighting the growing sophistication of malicious actors and the vulnerabilities present in smart contracts.

The trend continued into 2022, with total losses reaching an alarming $2.7 billion. This escalation underscores the urgent need for more robust security measures and the development of more secure smart contracts.

2.5.3. Chainalysis crypto crime reports

According to the Chainalysis 2021 Crypto Crime Report, the increase in financial losses due to smart contract exploits was part of a larger trend of rising cryptocurrency crime [11]. The report highlighted that scams, including those involving smart contracts, were the most significant contributors to cryptocurrency crime in 2020.

In the Chainalysis 2022 Geography of Cryptocurrency Report, it was noted that despite a tumultuous year in cryptocurrency and the onset of a bear market[10], cryptocurrency usage and adoption continued to grow around the world [14]. This growth, while beneficial for the

[10]A bear market refers to a condition in a financial market where the prices of securities are falling or are expected to fall. It is typically characterized by a prolonged period of decline in market prices, often by 20% or more from recent highs, and is accompanied by negative investor sentiment.

cryptocurrency ecosystem, also potentially increases the number of targets for smart contract exploits.

2.5.4. The need for enhanced security

The escalating financial losses due to smart contract exploits underscore the urgent need for enhanced security measures in the blockchain space. As the adoption of blockchain technology and cryptocurrencies continues to grow, so does the potential for financial loss due to smart contract exploits. This makes the development of secure smart contracts and the implementation of robust security measures a top priority for the blockchain community.

The work presented in this thesis contributes to this ongoing effort by developing a transformer-based model for detecting malicious smart contracts. By leveraging the power of machine learning and the vast amount of data available in the blockchain, this model aims to provide an effective tool for identifying and mitigating the risks associated with smart contract exploits.

In conclusion, the escalating trend of smart contract exploits and the associated financial losses highlight the significance and urgency of the work presented in this thesis. As the blockchain ecosystem continues to evolve, it is hoped that the development of robust security measures, such as the transformer-based model for detecting malicious smart contracts, will contribute to the safe and secure adoption of this transformative technology.

2.6. Machine learning for blockchain security

Blockchain technology, with its decentralized nature and cryptographic security, has revolutionized various sectors. However, the increasing complexity of smart contracts and the rapid growth of the blockchain ecosystem have also opened up new avenues for security threats that we discussed above. Traditional security measures often fall short in effectively detecting and mitigating these threats, necessitating the need for more advanced and proactive solutions. This is where machine learning comes into play.

Machine learning possesses the capability to derive valuable insights from extensive datasets. When it comes to open and publicly accessible blockchain data, individuals have the opportunity to construct and train machine learning models for various purposes, such as the

identification of suspicious or irregular activities on the blockchain, the categorization of whale addresses[11], and the extraction of emerging trends[12].

The combination of detection bots and machine learning proves to be a powerful alliance, as machine learning excels at promptly identifying Web3 security vulnerabilities. Here are three methods through which machine learning effectively fulfills this role:

- **Detect new threats with greater recall** – Machine learning has the capacity to identify both established and emerging threats by analyzing historical data. It accomplishes this by identifying common attributes and recognizing similar patterns in new data. For instance, machine learning can predict forthcoming phishing attempts by examining past phishing attacks and transaction patterns. Additionally, machine learning aids in streamlining the management of substantial datasets and facilitates the construction of high-performing machine learning models. One notable approach involves the use of autoencoders, which are neural networks capable of reducing high-dimensional data into a more concise representation, automating the feature extraction process and transforming raw data into numerical features[12].

- **Reduce alert fatigue by separating signal from noise** – Machine learning is proficient at identifying potential threats with a high degree of certainty, effectively reducing the prevalence of false alarms commonly associated with heuristic-based systems. It excels at extracting underlying patterns from data, differentiating between normal and anomalous activities. This is particularly crucial since an excessive number of alarms can overwhelm an operator's capacity to address them all. As operators struggle to keep up, they often resort to cherry-picking alerts, leading to the unfortunate consequence of overlooking genuine threats[12].

- **Produce actionable insights from large data more efficiently** – Machine learning can process large datasets and condense them, enabling algorithms and machine learning applications to generate actionable insights more efficiently. For example, machine learning can cluster and categorize similar wallet addresses based on their transaction behaviors. Additionally, it can identify clusters of addresses controlled by malicious entities. This empowers developers of detection bots to create more targeted logic capable of either disregarding or filtering out addresses with specific characteristics[12].

[11] Whale addresses refer to cryptocurrency addresses that hold a significant amount of a particular cryptocurrency, such as Bitcoin or Ethereum. These addresses typically contain a substantial portion of the total supply of the cryptocurrency and are associated with large-scale investors or entities known as "whales" in the cryptocurrency market.

[12] `https://forta.org/blog/three-ways-machine-learning-can-detect-web3-security-threats/` (visited on 2023-05-08)

The integration of machine learning into blockchain security signifies a paradigm shift from reactive to proactive security measures. By leveraging machine learning's predictive capabilities, we can anticipate potential threats before they materialize, thereby enhancing the security of the blockchain ecosystem.

In the subsequent chapters, we will delve deeper into the theory of transformers and their application in machine learning models for blockchain security. As we transition into this new era of blockchain security, it is crucial to remember that the ultimate goal remains the same - to create a secure, transparent, and efficient blockchain ecosystem for all.

3. Transformer-based Machine Learning Models for Smart Contract Analysis

3.1. Selection of machine learning method in security of smart contracts

In the realm of machine learning, numerous architectures and algorithms exist, each with their own strengths and limitations. When addressing the crucial task of detecting malicious smart contracts in real-time, choosing the most suitable machine learning architecture becomes paramount. Traditional methods like Support Vector Machines (SVMs) and Random Forests excel at handling structured data, but they lack the ability to process sequence information, which is intrinsic to smart contract data. Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks, on the other hand, can handle sequence information but may struggle with long sequences due to their sequential computation and issues with long-term dependencies.

As we delve into the exploration of machine learning architectures for this critical task, we inevitably encounter the revolutionary transformer architectures. The use of transformers in the realm of machine learning has been met with considerable acclaim, particularly in tasks involving sequence classification. Unlike RNNs and LSTMs, transformers do not process data sequentially; instead, they leverage self-attention mechanisms to simultaneously weigh the importance of different parts of the input sequence. This unique approach overcomes the limitations of sequential computation and long-term dependencies [4, 17].

The parallel computation ability of transformers not only enhances their efficiency but also positions them as excellent candidates for real-time applications, such as detecting malicious smart contracts. Furthermore, the self-attention mechanism equips transformers with the capability to capture relationships between different parts of the input sequence, regardless of their distance from each other. This is particularly crucial when dealing with smart contract data, where relationships between different parts of the contract are vital in determining its legitimacy.

Critically, recent research has addressed potential issues with transformers, such as the sequence length learning problem, where models use sequence length as a predictive feature instead of relying on important textual information. Such advances pave the way for more refined and robust transformer models for sequence classification tasks [15].

Justifying the focus on transformer architectures for this specific task, it becomes evident that their unique advantages set them apart from other models. Parallel computation ensures speed, allowing real-time performance, which is essential for timely detection of malicious smart contracts. Moreover, the attention mechanism grants transformers the ability to learn dependencies between data elements, regardless of their spatial separation, making them capable of handling long-range dependencies effectively.

While experimental verification of alternative methods can provide additional insights, the body of literature surrounding transformers and their demonstrated superiority in handling sequence data, efficiency, and performance make them the most promising candidate for real-time detection of malicious smart contracts. As we move forward in securing smart contracts, future research and development efforts should focus on optimizing transformer models and addressing their limitations to maximize their effectiveness.

3.2. Understanding transformer architecture

3.2.1. Origin and evolution

The transformer architecture, a groundbreaking innovation in the field of natural language processing (NLP) and machine learning, was first introduced by Vaswani et al. in the influential paper “Attention is All You Need” in 2017 [4]. This architecture proposed a novel approach to handling sequential data, focusing on the concept of self-attention mechanisms. Before the advent of transformers, Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs) were the primary architectures used for sequence modeling tasks. However, transformers have surpassed both RNNs and CNNs in various NLP benchmarks and have become the foundation of state-of-the-art models, such as BERT, GPT, and T5.

3.2.2. Key components and functioning

The transformer architecture is composed of two primary components: the encoder and the decoder. Both the encoder and the decoder consist of multiple layers of identical sub-modules, which include multi-head self-attention mechanisms, position-wise feed-forward networks, and layer normalization. The architecture also employs residual connections and positional encoding to infuse positional information into the model [4]. These components will be discussed in more detail in the following subsections.

The transformer architecture has proven to be highly effective for a wide range of NLP tasks, such as machine translation, sentiment analysis, text summarization, and question-answering. Its ability to capture long-range dependencies and complex relationships within sequential data makes it a promising candidate for analyzing smart contracts and detecting vulnerabilities in their code[1].

3.2.3. Multi-head self-attention mechanism

The multi-head self-attention mechanism is a core component of the transformer architecture. It allows the model to weigh the importance of different parts of the sequence when processing each element. This is particularly useful in the context of smart contract analysis, as it enables the model to focus on the most relevant opcodes when identifying potential vulnerabilities or malicious behavior [6].

Mathematically, the self-attention mechanism can be described as follows: Given a sequence of input vectors $X = (x_1, x_2, \ldots, x_n)$, the self-attention mechanism computes a weighted sum of the input vectors for each element in the sequence. The weights are determined by a softmax function applied to the dot product of the input vectors:

$$\text{Attention}(X) = \text{Softmax}\left(XX^T\right)X$$

In the multi-head self-attention mechanism, the input vectors are first transformed into multiple sets of vectors (or "heads"), each of which is processed independently by the self-attention mechanism. The outputs of all heads are then concatenated and linearly transformed to produce the final output:

$$\text{MultiHead}(X) = \text{Concat}\left(\text{head}_1, \text{head}_2, \ldots, \text{head}_k\right)W_O$$

where each head_i is computed as:

$$\text{head}_i = \text{Attention}\left(XW_{Qi}, XW_{Ki}, XW_{Vi}\right)$$

and W_{Qi}, W_{Ki}, and W_{Vi} are weight matrices for the query, key, and value transformations, respectively, and W_O is the output weight matrix.

3.2.4. Position-wise feed-forward networks

Position-wise Feed-Forward Networks (FFNs) are another key component of the transformer architecture. They consist of two linear transformations with a ReLU activation function

[1] https://kazemnejad.com/blog/transformer_architecture_positional_encoding/ (visited on 2023-05-14)

in between, and are applied independently to each position in the sequence [18]. This allows the model to learn complex patterns and representations for each opcode in the bytecode.

Mathematically, the FFN can be described as follows:

Given an input vector x, the FFN applies two linear transformations with a ReLU activation in between:

$$\mathrm{FFN}(x) = W_2 \,\mathrm{ReLU}\,(W_1 x + b_1) + b_2$$

where W_1, W_2, b_1, and b_2 are the weight matrices and bias vectors of the two linear transformations, respectively.

3.2.5. Layer normalization and residual connections

Layer normalization and residual connections are used in the transformer architecture to stabilize the training process and facilitate the learning of complex patterns. Layer normalization is applied to the outputs of both the multi-head self-attention and position-wise feed-forward networks. Given an input vector x, layer normalization computes the following:

$$\mathrm{LayerNorm}(x, \gamma, \beta, \epsilon) = \gamma \frac{x - \mathrm{Mean}(x)}{\sqrt{\mathrm{Variance}(x) + \epsilon}} + \beta$$

where γ and β are learnable scale and shift parameters, and ϵ is a small constant for numerical stability. Residual connections, on the other hand, are used to bypass each sub-module in the transformer architecture. This allows the model to learn more effectively by mitigating the vanishing gradient problem, which is a common issue in deep neural networks. Mathematically, a residual connection can be described as follows:

Given an input x and a function F representing a sub-module in the transformer (such as the multi-head self-attention or the FFN), the output y of the residual connection is computed as:

$$y = F(x) + x$$

This means that the output of the sub-module is added to the original input, allowing the model to learn a residual function with respect to the input [12].

3.3. BERT - Bidirectional Encoder Representations from Transformers

BERT, or Bidirectional Encoder Representations from Transformers, is a method of pre-training language representations that has significantly impacted the field of natural language processing (NLP). It has been instrumental in achieving state-of-the-art results on a variety of

NLP tasks. BERT was created and published in 2018 by researchers at Google AI Language. It's considered a significant leap in the field of machine learning for natural language processing. The model has been open-sourced and has been a key contributor to the rapid growth of NLP applications in recent years[2].

3.3.1. Architecture

BERT's architecture is based on the Transformer model, which uses an attention mechanism to understand the context of words in a sentence. The architecture consists of two main parts:

- **Embedding Layer** – This is the initial layer where words are converted into vectors that represent the words in a multi-dimensional space.
- **Transformer Blocks** – These are the core of BERT's architecture. Each block consists of two sub-layers: a multi-head self-attention mechanism and a position-wise fully connected feed-forward network.
  - **Self-Attention Mechanism** – This mechanism allows the model to focus on different words during training. It gives the model the ability to understand the context and semantics of a word based on its surrounding words.
  - **Feed Forward Neural Network** – This is a simple neural network that is applied to each position separately and identically[2].

3.3.2. Advantages and disadvantages

By exploring both BERT's strengths and limitations, we can gain a comprehensive understanding of this model. Let's start by examining the advantages of BERT:

1. **Bidirectional Contextual Encoding** – Unlike previous models, BERT takes into account the context from both the left and the right of a word in all layers. This allows for a deeper understanding of the word's context.
2. **Pre-training and Fine-tuning** – BERT is first pre-trained on a large corpus of text, then fine-tuned for specific tasks. This two-step process allows BERT to adapt to a wide variety of tasks with minimal task-specific parameters.
3. **State-of-the-art Performance** – BERT has achieved state-of-the-art results on a variety of NLP tasks, such as question answering and language inference.

Despite its advantages, BERT also has some drawbacks:

[2] `https://en.wikipedia.org/wiki/BERT_(language_model)` (visited on 2023-05-16)

1. **Computational Intensity** – BERT requires a significant amount of computational resources for training, which can be a barrier for those without access to high-end hardware.

2. **Requires Large Datasets** – To fully leverage its power, BERT needs to be trained on large datasets. This can be a challenge for tasks with limited available data.

3.3.3. Technical intricacies

BERT introduces several technical innovations:

– **Positional Encoding** – BERT uses positional encodings to inject information about the position of the words in the sequence. This allows the model to understand the order of the words. We will analyze this feature in the next section.

– **Masked Language Model (MLM)** – During training, 15% of the words in each sequence are replaced with a [MASK] token. The model then tries to forecast the initial value of the hidden words by using the context provided by the remaining, unhidden words in the sequence. This allows the model to understand the context of a word based on all of its surroundings (left and right of the word).

– **Next Sentence Prediction (NSP)** – BERT is also trained on a task where two sentences are input and the model must predict if the second sentence is the subsequent sentence in the original document. This helps the model understand the relationship between two sentences[2].

3.3.4. Positional encoding

Since the transformer architecture does not inherently capture the order of the elements in the input sequence, positional encoding is used to inject information about the position of each element in the sequence. This is particularly important in the context of smart contract analysis, as the order of the opcodes in the bytecode can significantly affect the behavior of the contract.

In the transformer architecture, positional encoding is achieved by adding a specific vector to each input element based on its position in the sequence. The positional encoding vector for position p and dimension i is computed as follows:

$$\mathrm{PE}(p,i) = \begin{cases} \sin\left(p/10000^{2i/d_{\text{model}}}\right) & \text{if } i \text{ is even} \\ \cos\left(p/10000^{2i/d_{\text{model}}}\right) & \text{if } i \text{ is odd} \end{cases}$$

where d_{model} is the dimensionality of the input vectors [7]. By adding these positional encoding vectors to the input vectors, the transformer model is able to take into account the order of the opcodes in the bytecode during its analysis.

3.3.5. DistilBERT

DistilBERT is a smaller, faster, and cheaper version of BERT. It was introduced by the team at Hugging Face as a way to mitigate some of the challenges associated with deploying BERT for real-world applications. Despite its reduced size and complexity, DistilBERT retains most of the performance of BERT, making it an ideal choice for tasks that require less computational resources [7].

The key to DistilBERT's efficiency is a process known as knowledge distillation. During training, a larger, more complex model (the teacher) is used to generate predictions for a dataset. These predictions, which capture the teacher model's understanding of the data, are then used to train a smaller, simpler model (the student). The student model learns to mimic the teacher's predictions, effectively distilling the teacher's knowledge into a more compact form.

In the case of DistilBERT, the teacher model is BERT, and the student model is a smaller version of BERT with fewer layers. The distillation process allows DistilBERT to learn a similar representation of the data as BERT, but with a fraction of the computational cost [7].

DistilBERT shares the same architecture as BERT, with a few key differences:

- DistilBERT has fewer Transformer layers. The base version of DistilBERT has 6 layers, compared to 12 in BERT-base.
- DistilBERT does not have token-type embeddings, which are used in BERT to distinguish between different sentences in the same input.
- DistilBERT uses sinusoidal position embeddings, as opposed to the learned position embeddings in BERT.

Despite these differences, DistilBERT performs remarkably well on a variety of NLP tasks. It achieves 97% of BERT's performance on the GLUE benchmark, while being 60% faster and 40% smaller in size [7].

3.4. Application of transformer models in smart contract analysis

Given the capabilities of transformer models in handling sequential data and capturing complex relationships, they are well-suited for the task of analyzing smart contracts. By training a transformer model on a dataset of smart contracts, we can equip it to detect patterns and relationships within the bytecode and opcodes of a contract, effectively learning to identify features that are indicative of malicious behavior or vulnerabilities.

For instance, the model could learn that certain opcode patterns are associated with common attack vectors, such as reentrancy attacks or integer overflows. By identifying these patterns in the bytecode, it becomes possible to flag potentially malicious contracts before they can be deployed or interact with other contracts on the blockchain.

Moreover, transformer-based models can be fine-tuned for specific tasks or domains, enabling them to be tailored to the unique characteristics of smart contract code and the Ethereum ecosystem. This adaptability makes them well-suited for the task of smart contract security analysis.

By using transformer-based models to analyze smart contracts, it becomes possible to proactively identify and flag potentially malicious contracts, even before they can be deployed or interact with other contracts on the blockchain. This early detection can help prevent attacks and minimize the potential damage caused by malicious actors.

In addition to identifying malicious contracts, transformer-based models can also be employed to analyze and detect vulnerabilities in legitimate contracts, enabling developers to address these issues before deployment. This proactive approach to smart contract security can help enhance the overall security and resilience of the blockchain ecosystem.

In the context of smart contract analysis, DistilBERT's efficiency and performance seem to make it a suitable choice for analyzing opcode sequences. Its ability to understand the context of each opcode, similar to how BERT comprehends words in a sentence, enables it to capture the intricate relationships and dependencies within the opcode sequences. This capability proves crucial in detecting malicious patterns and vulnerabilities.

4. Technical implementation

4.1. Tools utilized

– **Python**, a versatile and powerful programming language, served as the foundation. Its extensive ecosystem of libraries and frameworks facilitated the efficient development and implementation of complex machine learning models.

– The **Hugging Face** library, an invaluable natural language processing (NLP) toolkit, played a pivotal role. This comprehensive library offered a range of pre-trained models, tokenizers, and utilities, enabling exploration and experimentation with various NLP tasks.

– **PyTorch**, a widely adopted deep learning framework, proved essential for model training and evaluation. Its dynamic computational graph and user-friendly API facilitated the design and fine-tuning of neural networks, resulting in state-of-the-art results.

– **Forta**, a robust and innovative security monitoring solution, significantly enhanced the reliability and integrity within the Web3 ecosystem. With real-time threat detection capabilities, Forta enabled the identification and response to potential malicious activities.

– The **web3** library served as a fundamental component for seamless integration with blockchain technology. Leveraging web3, secure and transparent interaction with smart contracts allowed for efficient data storage and retrieval.

– **Docker**, a widely used containerization platform, played a crucial role in the deployment and usage of the bot.

4.2. Data collection and preprocessing

4.2.1. Data sources

The data used for training and evaluating the transformer-based model was collected from various sources. The primary source of data was a dataset published on HuggingFace[1] by the Forta team, which contains a diverse collection of smart contracts and their associated bytecode, opcodes, and metadata. To ensure the model's effectiveness in real-world scenarios, the dataset was expanded manually during the period of writing this thesis.

To address this need, an additional 115 smart contracts were randomly selected from those deployed during the period of writing this thesis and added to the dataset. This deliberate inclusion of recently deployed contracts broadened the range of smart contract implementations, thereby enhancing the diversity and representativeness of the dataset.

In total, the dataset consists of 115 malicious contracts, 10,000 benign contracts from the original dataset, and an additional 115 smart contracts deployed during the period of writing this thesis.

4.2.2. Data preprocessing

Once the data was collected, several preprocessing steps were carried out to prepare the smart contract data for training and evaluation with the transformer-based model:

– Data selection and retrieval: The original Forta dataset contained several columns, but only `'contract_address'` `(string)` and `'malicious'` `(bool)` were retained for the model. The `'contract_creation_tx'` `(string)` field was also used to retrieve opcodes before it was removed. A Python script was used to collect these opcodes:

```
from web3 import Web3

# Initialize Web3 with the Ethereum gateway URL
web3 = Web3(Web3.HTTPProvider("https://url-to-eth-gateway"))

addresses = []
```

[1] HuggingFace is a company and an open-source community that is widely known for its contributions to natural language processing (NLP) and machine learning. They have developed a popular platform and library that provides easy access to pre-trained NLP models, datasets, and other tools. Their platform allows researchers, developers, and students to quickly utilize and experiment with state-of-the-art NLP models without the need to train them from scratch.

```
opcodes = []

# Iterate over the contract data
for index in range(len(sc_df)):
    address = sc_df.contract_address[index]
    tx_hash = sc_df.contract_creation_tx[index]

    # Retrieve the bytecode of the contract
    code = web3.eth.getCode(
           Web3.toChecksumAddress(address)).hex()

    # If the bytecode is missing,
    # attempt to retrieve it from the transaction input
    if code == '0x':
        for _ in range(10):
            try:
                tx = web3.eth.getTransaction(tx_hash)
                code = tx.input
                if code == b'':
                    raise Exception
                break
            except:
                continue

    # If no bytecode is available, skip to the next contract
    if not code:
        continue

    # Disassemble the bytecode into opcodes
    opcode = EvmBytecode(code).disassemble()

    # Initialize a string to store the
    # formatted opcode sequence
    result = ''

    # Iterate over the opcodes and format them
    for j in opcode:
```

```
            result += f'{j.name} '
            if j.operand:
                result += f'0x{j.operand} '

        # Remove the trailing space and append
        # the formatted opcode sequence to the list
        result = result[:-1]
        addresses.append(address)
        opcodes.append(result)
```

– Tokenization: The smart contract opcodes were tokenized to convert them into a format that can be fed into the transformer-based model. This involved breaking the bytecode and opcodes into individual tokens or sub-tokens, which represent the smallest units of meaning within the code.

```
def tokenize_function(dataset):
    return tokenizer(dataset["opcodes"], truncation=True)

# Apply the tokenize function to the raw dataset
tokenized_datasets = raw_dataset.map(
                     tokenize_function, batched=True)

# Remove the 'opcodes' column from the tokenized dataset
tokenized_datasets = tokenized_datasets.\
                     remove_columns('opcodes')
```

– Data split: The preprocessed dataset was split into training and test sets, ensuring an appropriate distribution of data samples across these sets. This allowed for the model to be trained on a diverse range of smart contracts and test on unseen data to measure its generalization performance.

```
# Split the raw dataset into training and testing datasets
raw_dataset = raw_dataset.train_test_split(0.1, 0.9, seed=42)
```

– Sequence padding: Due to the varying lengths of smart contract bytecode and opcode sequences, it was necessary to pad the input sequences to ensure a consistent input size for the model. This was achieved by adding padding tokens to the end of shorter sequences until they reached the predetermined maximum sequence length.

```
# Load the pre-trained DistilBert tokenizer
tokenizer = DistilBertTokenizerFast.from_pretrained(
            "distilbert-base-uncased")

# Create a data collator that will be used
# to pad the inputs to the same length
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)

# Create a DataLoader for the training dataset
# This will shuffle the data and batch it, using the
# data collator to pad inputs to the same length
train_dataloader = DataLoader(
    tokenized_datasets["train"], shuffle=True, batch_size=16,
    collate_fn=data_collator
)

# Create a DataLoader for the evaluation dataset
# This will batch the data, using the data collator
# to pad inputs to the same length
eval_dataloader = DataLoader(
    tokenized_datasets["test"], batch_size=16,
    collate_fn=data_collator
)
```

By carefully collecting and preprocessing the smart contract data, it was possible to create a robust and representative dataset for training and evaluating the transformer-based model. The inclusion of more recent smart contracts, in particular, helped address the limitations observed in the initial experiments using the HuggingFace dataset alone, leading to improved precision and overall performance.

4.3. Model training and evaluation

4.3.1. Model configuration and initialization

The model used for this task is a DistilBert model, which is a smaller and faster variant of the BERT model. It was chosen for its balance between performance and computational efficiency. The model was initialized with the pre-trained weights from the "distilbert-base-uncased" model, which has been trained on a large corpus of English text [7].

The model was configured for sequence classification, making it suitable for the binary classification task of identifying malicious smart contracts. The model's output logits were passed through a cross-entropy loss function, which is commonly used for binary and multi-class classification tasks.

Here is the code snippet for model initialization:

```
from transformers import DistilBertForSequenceClassification

model = DistilBertForSequenceClassification
        .from_pretrained("distilbert-base-uncased")
```

4.3.2. Data loading and tokenization

The data was loaded using the HuggingFace's `load_dataset` function, which provides a convenient way to load and preprocess datasets. The raw dataset was then tokenized using the DistilBert tokenizer. Tokenization is the process of converting text into tokens, which are numerical representations that the model can understand.

In the case of DistilBert, the tokenizer splits the input into subwords, or tokens, that are found in its vocabulary. For example, a word like "contract" might be split into "con", "##tra", "##ct". This process allows the model to handle a wide range of input while keeping the vocabulary size manageable.

The tokenized data was then organized into batches using PyTorch's DataLoader, which provides an efficient way to iterate over the data during training. The data was batched with a batch size of 16, and padding was applied to ensure that all sequences in a batch have the same length.

Here is the code snippet for data loading and tokenization:

```
from transformers import DistilBertTokenizerFast
from torch.utils.data import DataLoader
from transformers import DataCollatorWithPadding

tokenizer = DistilBertTokenizerFast
            .from_pretrained("distilbert-base-uncased")

def tokenize_function(example):
    return tokenizer(example["opcodes"], truncation=True)

data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

```
tokenized_datasets = raw_dataset.map(tokenize_function,
                                     batched=True)
train_dataloader = DataLoader(
    tokenized_datasets["train"], shuffle=True, batch_size=16,
    collate_fn=data_collator
)
```

4.3.3. Training procedure

The model was trained using the AdamW [5] optimizer. Here are the key reasons:

– **Weight decay**, also known as L2 regularization, is a technique that adds a penalty term to the loss function, encouraging the model to have smaller weight values. This regularization helps prevent overfitting by reducing the complexity of the model. The AdamW optimizer incorporates weight decay directly into its update rule, making it convenient to apply regularization without additional complexity.

– **Compatibility with Transformer Architectures** – The AdamW optimizer has been widely used and extensively tested in various deep learning applications, including transformer-based models. It has shown excellent performance in optimizing large-scale models with numerous parameters, such as transformers. The compatibility of the AdamW optimizer with transformers makes it a suitable choice for this specific model.

The initial hyperparameters were chosen according to the developers' recommendations [16], but later, some of the specified hyperparameters were adapted as a result of numerous experiments. The obtained results confirm that the hyperparameters were chosen correctly.

The learning rate was set to $5e-5$, and a linear learning rate scheduler was used to gradually decrease the learning rate over the course of training. The model was trained for 5 epochs instead of the recommended 3, which proved to be a more optimal choice in the experiments, where an epoch is one complete pass through the entire training dataset. During each epoch, the model's parameters were updated to minimize the loss on the training data. The loss function used was the cross-entropy loss, which is suitable for binary and multi-class classification tasks. The number of warmup steps was set to 600, as recommended for this model [16]. CrossEntropyLoss was used with weights as one of the methods to handle a highly imbalanced dataset. This way, false negative classification is $2x$ more "costly" for the model than false positive classification.

After each epoch, the model's performance was evaluated on the test data. This allowed for monitoring the model's generalization performance and early stopping if the model started to overfit.

Here is the code snippet for the training procedure:

```
from transformers import get_scheduler
from torch.optim import AdamW
from torch import nn
from tqdm.auto import tqdm

# Initialize the AdamW optimizer with the model parameters
# and learning rate
optimizer = AdamW(model.parameters(), lr=5e-5, weight_decay=0.01)

# Set the number of epochs and calculate the total
# number of training steps
num_epochs = 5
num_training_steps = num_epochs * len(train_dataloader)

# Get a learning rate scheduler that linearly decreases the learning
# rate from the initial lr set in the optimizer to 0
lr_scheduler = get_scheduler(
    "linear",
    optimizer=optimizer,
    num_warmup_steps=600,
    num_training_steps=num_training_steps,
)

# Define the loss function as CrossEntropyLoss and set the weight
# for each class
loss_fn = nn.CrossEntropyLoss(weight=torch.tensor([1.0, 2.0])\
          .to(device))

train_losses = []
val_losses = []

# Start the training loop
for epoch in range(num_epochs):
    # Initialize the total loss for this epoch
    loss_total = 0
    loss_val = 0
```

```
    # Set the model to training mode
    model.train()

    # Iterate over each batch in the training DataLoader
    for batch in tqdm(train_dataloader):
        # Move the batch tensors to the same device as the model
        batch = {k: v.to(device) for k, v in batch.items()}

        # Forward pass: compute the model outputs based on the
        # input batch
        outputs = model(**batch)
        logits = outputs.logits

        # Compute the loss value for this batch
        loss = loss_fn(logits.view(-1, 2), batch["labels"]\
               .view(-1))

        # Backward pass: compute the gradients of the loss
        loss.backward()

        # Update the model's parameters
        optimizer.step()

        # Update the learning rate
        lr_scheduler.step()

        # Reset the gradients
        optimizer.zero_grad()

        # Accumulate the total loss
        loss_total += loss.item()

    train_loss = loss_total / len(train_dataloader)
    train_losses.append(train_loss)
```

```
    # Set the model to evaluation mode
    model.eval()

    # Iterate over each batch in the evaluation DataLoader
    for batch in eval_dataloader:
        # Move the batch tensors to the same device as the model
        batch = {k: v.to(device) for k, v in batch.items()}

        # Forward pass: compute the model outputs
        # based on the input batch
        with torch.no_grad():
            outputs = model(**batch)
        logits = outputs.logits

        # Compute the loss value for this batch
        loss = loss_fn(logits.view(-1, 2), batch["labels"].view(-1))

        # Accumulate the total validation loss
        loss_val += loss.item()

    val_loss = loss_val / len(eval_dataloader)
    val_losses.append(val_loss)
```

4.3.4. Model evaluation

The model's performance was evaluated on a separate test set after the training phase. The evaluation metrics used were Accuracy, Precision, Recall, F1-score, and Specificity. These are standard metrics for binary classification tasks.

The model achieved the following results:

- Accuracy: 0.9961
- Precision: 0.7143
- Recall: 1.0
- F1-score: 0.8333
- Specificity: 0.9960

The high accuracy and recall indicate that the model was able to correctly identify most of the malicious smart contracts in the test set. The precision, while lower, is still satisfactory and can be improved with further tuning of the model. The F1-score, which is the harmonic mean of precision and recall, is also quite high, indicating a good balance between the two. The high specificity shows that the model is good at identifying the true negatives, i.e., the benign smart contracts.

The model's loss during the training phase was also plotted to visualize its learning progress. The plot shows how the loss decreased over time, indicating that the model was learning from the data and improving its predictions. The resulting loss is shown in the figure 4.1.

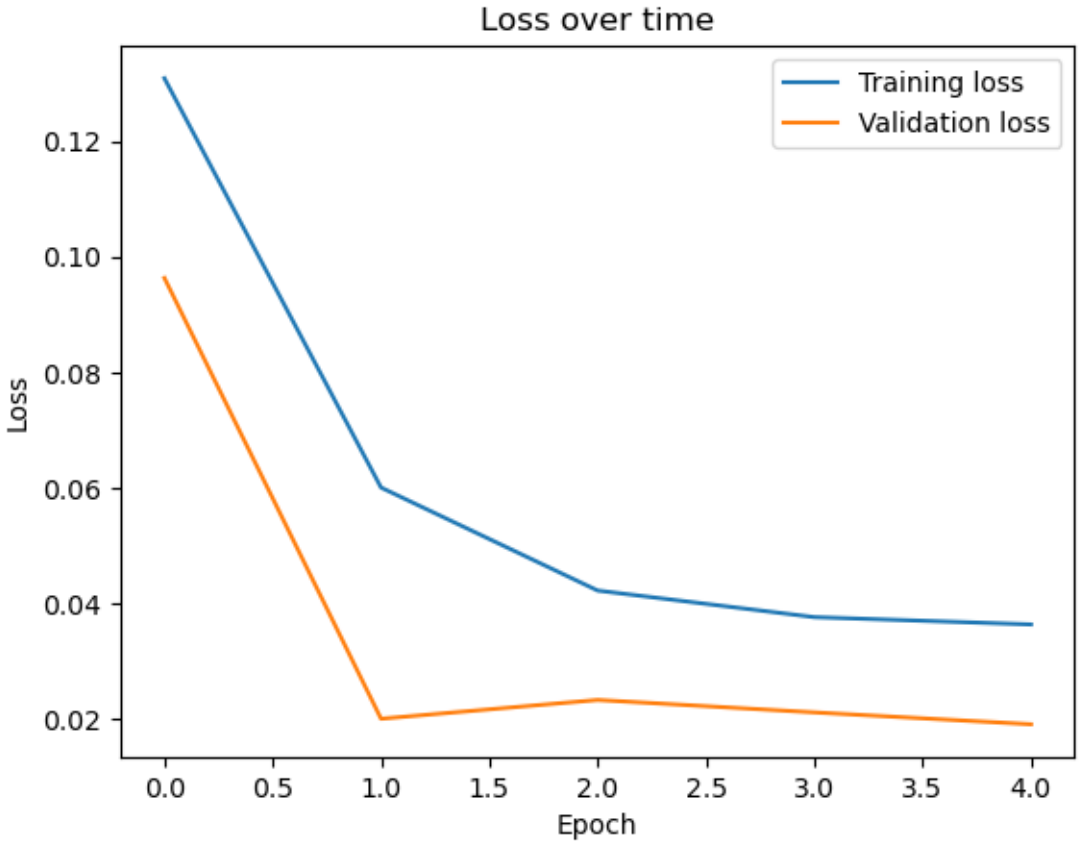

Fig. 4.1. Loss over time

Finally, a heatmap was created to visualize the model's predictions. The heatmap provides a graphical representation of the data where individual values are represented as colors. It helps in understanding the concentration and distribution of predictions across different classes. The resulting heatmap is shown in the figure 4.2.

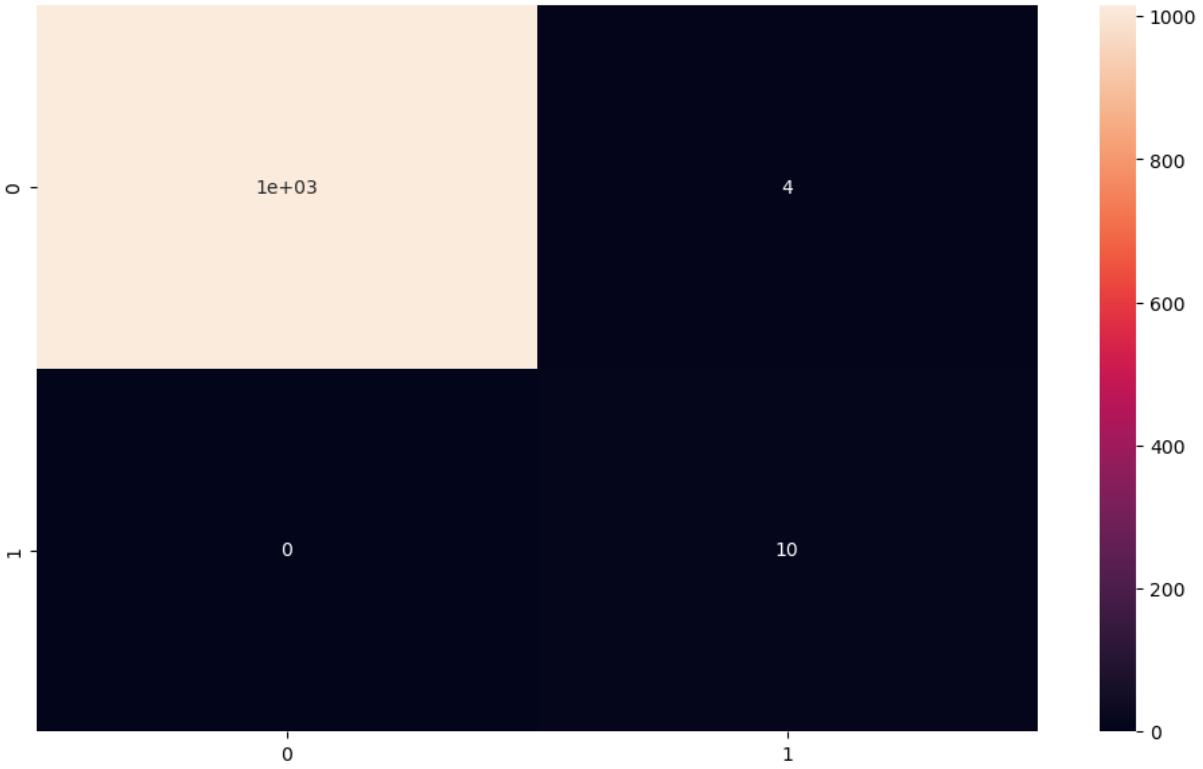

Fig. 4.2. Heatmap of model predictions

In conclusion, the model demonstrated high performance in detecting malicious smart contracts, as evidenced by the evaluation metrics and visualizations.

4.3.5. Model deployment and usage

The model is deployed and used in a Python environment, leveraging the Hugging Face Transformers library for the DistilBERT model and the PyTorch library for tensor computations. The model is used to analyze Ethereum transactions and detect potentially malicious smart contracts. For a visual representation of this detection flow, please refer to figure 4.3.

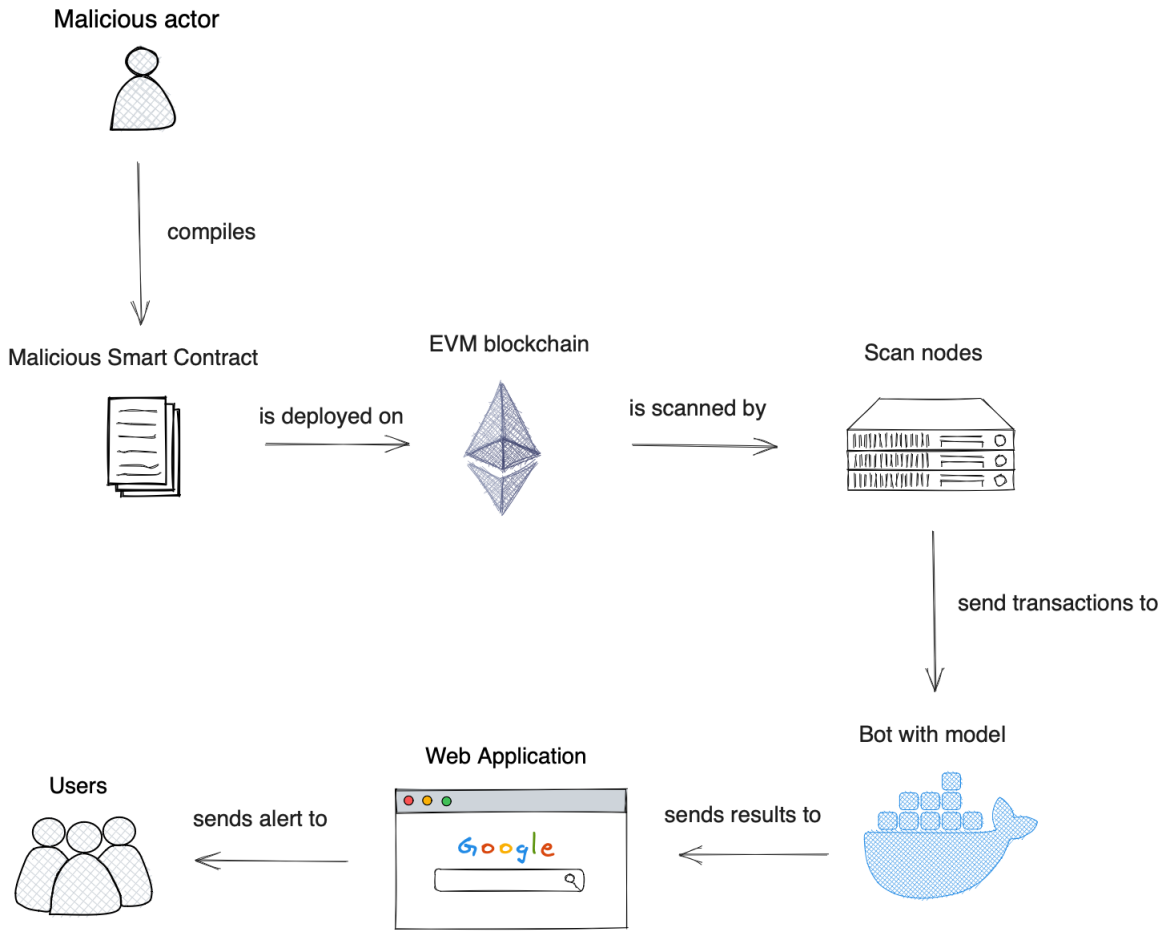

Fig. 4.3. General Detection Flow

The process begins with the initialization of the model. The pretrained DistilBERT model is loaded from the Hugging Face model hub, and the model weights are loaded from a local file. The model is then set to evaluation mode, which disables certain features like dropout that are used during training but not during inference. For a visual representation, please refer to figure 4.4.

The main function for detecting malicious contracts is $detect\_malicious\_contract\_tx$. This function takes as input a TransactionEvent object, which represents an Ethereum transaction, and returns a list of findings. The function first checks if the transaction includes any contract creation traces. If it does, it loops over these traces and for each one, it calls the $detect\_malicious\_contract$ function.

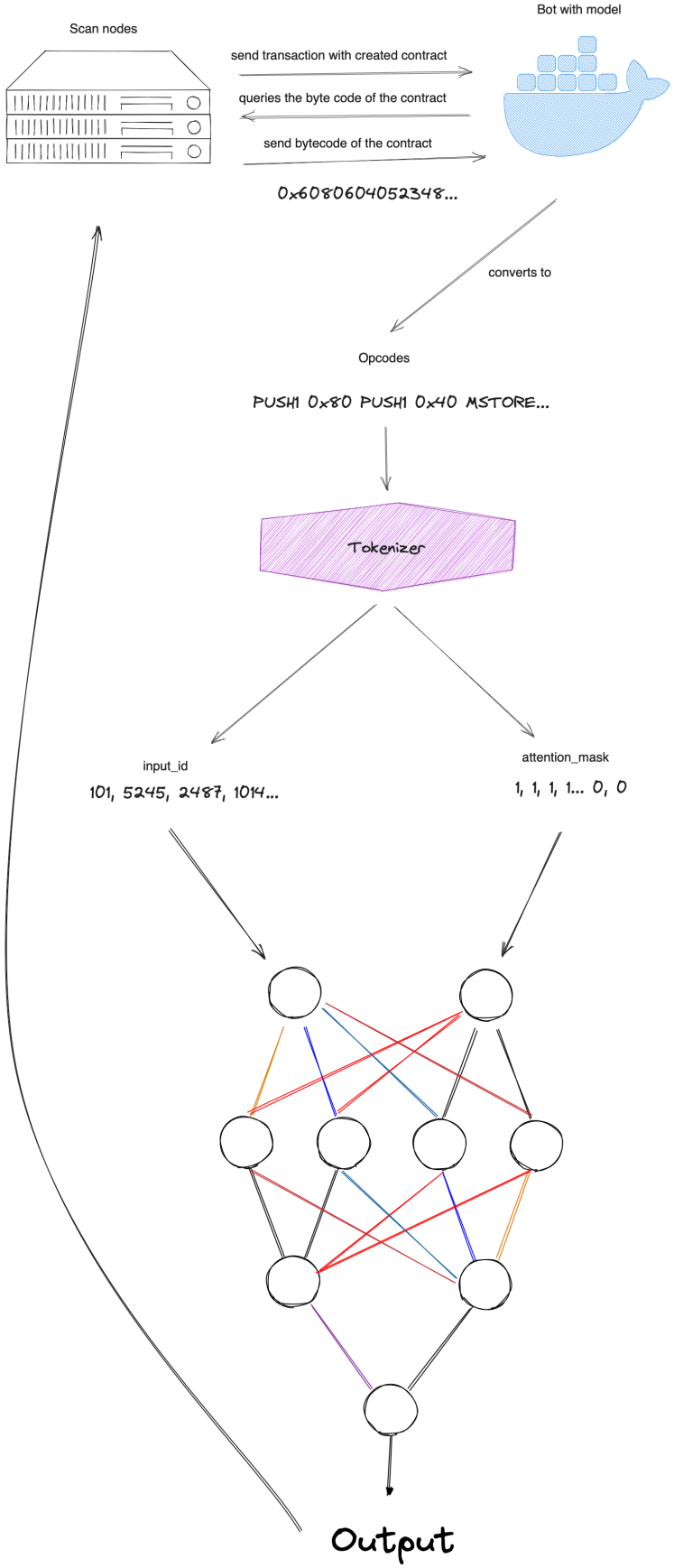

Fig. 4.4. General Application Flow

The *detect_malicious_contract* function takes as input the address of the contract creator and the address of the created contract. It retrieves the bytecode of the created contract from the Ethereum network, disassembles it into opcodes, and tokenizes the opcodes using the DistilBERT tokenizer. The tokenized opcodes are then fed into the DistilBERT model, which outputs a prediction. If the prediction indicates that the contract is malicious, a finding is created and added to the list of findings.

The *handle_transaction* function is the entry point for the agent. It takes as input a TransactionEvent object and passes it to the *detect_malicious_contract_tx* function. The findings returned by this function are then returned by the *handle_transaction* function.

Here is a simplified version of the code for better understanding:

```
import torch as tc
from transformers import DistilBertTokenizerFast
from transformers import DistilBertForSequenceClassification
from web3 import Web3
from evmdasm import EvmBytecode

# Initialize the model and tokenizer
model = DistilBertForSequenceClassification\
        .from_pretrained("distilbert-base-uncased")
model.load_state_dict(tc.load(
                        "./src/opcodes_model_weights_v5.pth"))
tokenizer = DistilBertTokenizerFast\
            .from_pretrained("distilbert-base-uncased")
model.eval()

# Connect to the Ethereum network
w3 = Web3(Web3.HTTPProvider(get_json_rpc_url()))

def detect_malicious_contract(from_, created_contract_address):
    # Get the contract bytecode and disassemble it into opcodes
    bytecode = w3.eth.get_code(
                Web3.toChecksumAddress(
                     created_contract_address)).hex()
    opcode = EvmBytecode(bytecode).disassemble()
    opcodes = ' '.join([f'{j.name} 0x{j.operand}'
              if j.operand else j.name for j in opcode])
```

```
    # Tokenize the opcodes and feed them into the model
    inputs = tokenizer([opcodes],
                       padding="max_length", truncation=True)
    input_ids = tc.tensor(inputs["input_ids"][0])
    attention_mask = tc.tensor(inputs["attention_mask"][0])
    with tc.no_grad():
        outputs = model(input_ids.unsqueeze(0),
                  attention_mask.unsqueeze(0))

    # If the model predicts the contract to be malicious,
    # create a finding
    if np.argmax(outputs[0].numpy()) == 1:
        return [MaliciousContractFindings.\
        malicious_contract_detected(from_, created_contract_address)]
    else:
        return []

def handle_transaction(transaction_event):
    findings = []
    # Check if the transaction includes any contract creation traces
    if len(transaction_event.traces) > 0:
        for trace in transaction_event.traces:
            if trace.type == "create":
                created_contract_address = trace.result.address
                if trace.result else None
                if created_contract_address is not None:
                    # Detect if the created contract is malicious
                    findings.extend(detect_malicious_contract(
                    trace.action.from_, created_contract_address))
    return findings
```

5. Experiments

This section explores two specific attacks that occurred in the DeFi space since the bot was enabled on April 10, 2023. The attacks discussed are an attack on Hundred Finance and a flash loan attack on Jimbo's Protocol. The analysis includes an overview of each attack, the execution process, the aftermath, and the lessons learned.

5.1. Attack on Hundred Finance

5.1.1. Overview

On April 15th, 2023, the bot detected a potential attack on Hundred Finance, which is a Compound fork that utilizes hTokens to track lending positions. The bot's alert was generated at 13:15:58 GMT[1], almost an hour before the actual attack occurred at 14:12:00 GMT[2]. This early detection theoretically provided ample time to prevent the attack.

5.1.2. Attack Execution

Hundred Finance suffered a significant exploit on the Optimism network, resulting in a loss of $7.4 million. Notably, this was not the first time Hundred Finance had been targeted. In February, the protocol lost 3.3 million when Meter was exploited, and in March, it lost $6.2 million in a dual-edged attack that also affected Agave DAO for $5.5 million.

The April attack was executed using a flashloan[3] of WBTC from Aave. The attacker donated large amounts of WBTC to an empty hWBTC contract, manipulating the exchange rate

[1] `https://explorer.forta.network/alert/0x3c9b919e24b2daa85d0fe2293f09b4ce38c3de27030c877eb2a77dbb91ba3e7d` (visited on 2023-05-28)

[2] `https://rekt.news/hundred-rekt2/` (visited on 2023-05-28)

[3] A flashloan attack is a type of exploit that takes advantage of the unique features of flashloans in decentralized finance (DeFi) platforms. Flashloans allow users to borrow a large amount of cryptocurrency without collateral as long as the borrowed funds are returned within the same transaction. In a flash loan attack, an attacker exploits vulnerabilities in a smart contract to execute a series of complex transactions within a single block, leveraging the borrowed funds to manipulate prices, exploit arbitrage opportunities, or cause disruptions in the targeted protocol.

between hWBTC and WBTC. A rounding error in the `redeemUnderlying` function further facilitated the exploit[2].

5.1.3. Aftermath

Following the attack, the hacker moved most of the stolen funds to Ethereum, where they were swapped for centralized stablecoins or deposited into Curve. The hacker's Debank profile showed approximately $5.4 million of assets on Ethereum and $0.9 million remaining on Optimism.

The price of the HND token dropped around 50% over the day following the hack. It has since recovered somewhat, reaching around $0.025, down from around $0.039 before the attack[2].

5.1.4. Lessons Learned

This incident highlights the vulnerabilities inherent in forking code. When one fork falls, all others must assess their foundations. In response to the attack, Hundred Finance advised other COMP forks to reach out, warning that the hack exploited "a general flaw in the code and not specific to Hundred deployment." The Hundred team has also announced a reward for information leading to the identification of the hacker, demonstrating their commitment to addressing this issue and preventing future attacks[2].

5.2. Flash Loan Attack on Jimbo's Protocol

5.2.1. Overview

This section discusses the flash loan attack that targeted Jimbo's Protocol, resulting in a significant loss of $7.5 million. The attack took place on May 28, 2023 at 00:40:28 GMT[4]. The presence of the malicious contract was initially detected by the bot at 00:18:51 GMT on the same day[5].

Flashloan attacks have been responsible for significant losses in the DeFi space and highlight the importance of robust security measures and code auditing in decentralized applications.

[4] `https://rekt.news/jimbo-rekt/` (visited on 2023-06-14)

[5] `https://explorer.forta.network/alert/0xcdc59c93dc4b71a83503d2b5401544a98efb97239dd4299bc64a27126ccd456d` (visited on 2023-06-14)

5.2.2. Attack Execution

Jimbo's Protocol, an innovative project aiming to create a semi-stablecoin through rebalancing mechanisms, had recently undergone a relaunch with its v2 version. This new iteration was developed to address the shortcomings of the previous v1 version.

Concerns about the complexity of Jimbo's Protocol were raised, as the relaunch of the v2 version incorporated leverage and introduced additional complexity, raising questions about the feasibility of such a multifaceted decentralized finance (DeFi) protocol.

Shortly after three days of operating under the v2 protocol, the flash loan attack unfolded. Although the team behind Jimbo's Protocol was alerted via Twitter, it took them approximately six hours to publicly acknowledge the incident[4].

5.2.3. Aftermath

Insights provided by security firms highlighted that the exploit was attributed to a lack of slippage control in the `shift` function of the JimboController contract.

To execute the attack, the perpetrator leveraged a flashloan of 10k ETH, artificially inflating the price of JIMBO tokens. By strategically depositing overvalued JIMBO into the JimboController, the attacker triggered a rebalance through the shift() function, leading to the transfer of the contract's WETH back into the pool. As a result, the attacker liquidated the remaining JIMBO tokens, depleting the WETH liquidity and causing a significant crash in the price of JIMBO.

The attacker's address associated with the incident is 0x102be4bccc2696c35fd5f5bfe54c1dfba416a741, while the stolen funds, totaling over 4000 ETH ($7.5M), were moved to another address on the Ethereum network.

In an attempt to recover the stolen funds, Jimbo's Protocol has engaged directly with the attacker on-chain, offering a 10% bounty for the return of the funds. Expressing their determination, they have made it clear that if the matter remains unresolved, they will collaborate with law enforcement agencies[4].

6. Conclusion

6.1. Achieved goals

The primary objective of this thesis was to explore the application of transformer-based machine learning models for detecting malicious smart contracts. This objective has been successfully achieved. The thesis provides a comprehensive overview of the Ethereum platform and the security concerns related to smart contracts. It further examines the intricacies of transformer models, particularly the Bidirectional Encoder Representations from Transformers (BERT), and their effectiveness in the context of smart contract analysis.

A model was developed that can effectively identify potential security threats in the code of smart contracts, thereby contributing to the enhancement of security in the blockchain ecosystem. The model was trained and evaluated using real-world data, and its performance was found to be promising. The thesis also presented a real-world attack detected by the bot, providing valuable insights into the practical application of the model.

6.2. Results discussion and limitations

While labeling a smart contract as malicious is a crucial step in identifying potential threats, it is not sufficient to prevent an attack. This is due to the complex nature of blockchain ecosystems and the sophisticated strategies employed by attackers.

6.2.1. Noise in the blockchain ecosystem

In reality, multiple contracts are created on various blockchains every day. Some of these contracts may contain malicious patterns, creating a significant amount of noise that makes it difficult to identify a real attack. This is further complicated by the fact that not all contracts with malicious patterns are used for malicious purposes. Therefore, relying solely on the identification of malicious patterns in smart contracts is not enough to effectively prevent attacks.

6.2.2. The need for additional indicators

To distinguish a real threat from noise, it is necessary to track other signs of an attack. For instance, the creation of a new account and the receipt of funds from anonymizing sources like mixers[1] could indicate a potential attack. There are numerous such patterns, and only collectively can we distinguish a real threat from noise.

6.2.3. Response strategy

In addition to identifying potential threats, it is also important to develop a response strategy to inform developers and users about potential dangers. This could involve alerting the relevant parties, providing detailed information about the threat, and offering guidance on how to mitigate the risk.

6.3. Future development

While the results achieved are encouraging, there is ample scope for further development. The model can be refined and improved with the incorporation of more data and the use of more advanced machine learning techniques. The scope of the model can also be expanded to include smart contracts developed in other programming languages and deployed on other blockchain platforms.

Furthermore, the thesis highlighted the need for additional indicators to distinguish real threats from noise in the blockchain ecosystem. Future work could focus on identifying and incorporating these indicators into the model to enhance its effectiveness. The importance of a response strategy was also underscored. Future efforts could be directed towards developing a robust response strategy to inform developers and users about potential dangers in a timely and effective manner.

In conclusion, the application of transformer-based machine learning models in the analysis and detection of smart contract vulnerabilities holds considerable promise. With further

[1]Mixers, in the context of blockchain and Ethereum, refer to privacy-enhancing mechanisms designed to increase the anonymity and fungibility of digital assets. Mixers are used to obfuscate the transaction history and break the linkability between sender and recipient addresses.

When users participate in transactions on the Ethereum blockchain, the transaction details, including the sender's and recipient's addresses, are publicly visible on the blockchain. This transparency raises privacy concerns, as anyone can analyze the blockchain and trace the flow of funds.

To address this issue, mixers introduce a process where users can "mix" their funds with other participants' funds. The mixing process involves pooling funds from multiple participants into a single transaction, effectively blending them together. The mixed funds are then redistributed to the participants' designated addresses.

research and development, it can play a crucial role in enhancing security in the rapidly evolving blockchain ecosystem.

Bibliography

[1] Nick Szabo. *Smart Contracts*. 1994. URL: *https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html* (visited on 2023-04-23).

[2] Vitalik Buterin. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. 2013. URL: *https://ethereum.org/en/whitepaper/* (visited on 2023-04-12).

[3] Gavin Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. Ethereum. 2014. URL: *https://ethereum.github.io/yellowpaper/paper.pdf* (visited on 2023-04-12).

[4] Niki Parmar Jakob Uszkoreit Llion Jones Aidan N. Gomez Lukasz Kaiser Illia Polosukhin Ashish Vaswani Noam Shazeer. *Attention Is All You Need*. 2017. URL: *https://arxiv.org/abs/1706.03762* (visited on 2023-04-18).

[5] Frank Hutter Ilya Loshchilov. "Decoupled Weight Decay Regularization". In: *ArXiv* abs/1711.05101 (2018).

[6] F. Moiseev Rico Sennrich Ivan Titov Elena Voita David Talbot. "Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned". In: (2019). DOI: *10.18653/v1/P19-1580*.

[7] Victor Sanh et al. *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. 2019.

[8] Zibin Zheng et al. "An Overview on Smart Contracts: Challenges, Advances and Platforms". In: (2019). `http://arxiv.org/pdf/1912.10370`.

[9] G. A. Oliva, A. Hassan, and Z. Jiang. *An exploratory study of smart contracts in the Ethereum blockchain platform*. 2020. URL: *https://dx.doi.org/10.1007/s10664-019-09796-5* (visited on 2023-04-23).

[10] Kaihua Qin et al. *Attacking the DeFi Ecosystem with Flash Loans for Fun and Profit*. 2020. URL: *http://arxiv.org/pdf/2003.03810* (visited on 2023-04-25).

[11] *2021 Crypto Crime Report*. Chainalysis. 2021.

[12] Futao Wei Yuxuan Lou Yong Liu Yang You Fuzhao Xue Ziji Shi. "Go Wider Instead of Deeper". In: (2021). DOI: *10.1609/aaai.v36i8.20858*.

[13] Satpal Singh Kushwaha et al. *Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract*. 2021. URL: *https://dx.doi.org/10.1109/ACCESS.2021.3140091* (visited on 2023-04-23).

[14] *2022 Geography of Cryptocurrency Report*. Chainalysis. 2022.

[15] Jean-Thomas Baillargeon and Luc Lamontagne. *Reducing Sequence Length Learning Impacts on Transformer Models*. 2022. URL: *https://arxiv.org/abs/2212.08399* (visited on 2023-05-12).

[16] HF Canonical Model Maintainers. *distilbert-base-uncased-finetuned-sst-2-english (Revision bfdd146)*. 2022. DOI: *10.57967/hf/0181*.

[17] Abhishek Gupta Raunak Joshi. *Performance Comparison of Simple Transformer and Res-CNN-BiLSTM for Cyberbullying Classification*. 2022. URL: *https://arxiv.org/abs/2206.02206* (visited on 2023-05-12).

[18] Xianchao Wu. "Deep Sparse Conformer for Speech Recognition". In: (2022). DOI: *10.21437/interspeech.2022-10384*.

[19] *2023 Crypto Crime Report*. Chainalysis. 2023.