

THREAT PREVENTION & VULNERABILITY ANALYSIS OF SMART CONTRACTS IN BLOCKCHAIN NETWORKS

Pankaj Chandra*, Santosh Soni, Akanksha Gupta, Prayas Kumar, Kunal Raj

Received: May 15, 2023; Revised: June 26, 2023; Accepted: September 18, 2023

Abstract

Blockchain technology now relies heavily on smart contracts, which offer self-executing code for the exchange of goods and data. Smart contracts enable financial activities like payments and auctions on blockchain systems. Even if bugs are found, they cannot be changed. They are frequently used to automate the implementation of agreements so that all parties may be confident in the conclusion without the need for an intermediary. But because of their openness and how they interact with one another in the blockchain ecosystem, these contracts are vulnerable to security risks. Smart contract security research has advanced significantly in recent years. The research community of smart contract security has made a significant improvement recently. Researchers have identified several security flaws in smart contracts and have created frameworks for verification and static analysis to find them. This study attempts to identify the most critical software security issues that affect smart contracts. These are then handled utilizing a variety of methods and equipment common to the sector.

Keywords: Smart contract auditing, Delegate call, Self destruct, Denial of Service, Time-stamp Manipulation, Reentrancy

Introduction

Due to its decentralized and secure nature, blockchain technology is quickly gaining favor. The smart contract, a self-operating piece of code, which runs automatically when certain circumstances are satisfied, is one of the main elements of this technology. Within the blockchain context, these contracts are designed to streamline transactions and the exchange of assets and information. Millions of

dollars are stored in Ethereum smart contracts, so ensuring that they are executed correctly is essential to preventing theft-related threats (Luu *et al.*, 2016). Using smart contracts had a string of security problems that cost the company a lot of money (Atzei *et al.*, 2017). In order to identify potential security vulnerabilities in smart contract source code, we offered numerous security assurance

Department of Information Technology, School of Studies in Engineering and Technology, Guru Ghasidas Vishwavidyalaya, Bilaspur, Chhattisgarh, India - 495009. E-mail: pankaj2684@gmail.com

* Corresponding author

DOI: <https://doi.org/10.55766/sujst-2023-05-e01234>

Suranaree J. Sci. Technol. 30(5):010256(1-19)

methods and methodologies in this study (Wang *et al.*, 2019). A smart contract's immutability and honest execution of transactions are guaranteed by blockchain technology. Ethereum, being the most well-known blockchain system that allow smart contracts. Despite their numerous advantages, smart contracts are subject to several security risks and flaws that might jeopardize their functionality and cause financial losses (Atzei *et al.*, 2017; Nikolic *et al.*, 2018).

In this study paper, we'll look at the most prevalent security risks that smart contracts must deal with as well as any potential flaws that may result from their use. Researchers have established different approaches to detect and reduce these security concerns, such as comparing historical versions (Chen, 2020a), conducting security analysis (Polishchuk *et al.*, 2012), using static

analysis (Ye *et al.*, 2020), creating security assessment frameworks (Jain *et al.*, 2022), identifying potential risks (Yamashita *et al.*, 2019), and conducting surveys (Liu and Liu 2019; Chen *et al.*, 2020b; Wan *et al.*, 2021; Kushwaha *et al.*, 2022). Additionally, some studies have focused on empirical analysis (Luu *et al.*, 2016; Hwang and Ryu, 2020) or specific vulnerabilities, such as integer bugs (Torres *et al.*, 2018). To ensure the safety of smart contracts, it is essential to consider these different approaches and apply appropriate techniques to address specific vulnerabilities. There are various practical security analysis tools for smart contracts (Perez and Livshits, 2021). Threats to Smart Contracts: There are a variety of security risks that smart contracts are susceptible to, including but not limited to (Table 1).

Table 1. Types of Security Risks

Vulnerability	Description	Additional Parameters
1. Reentrancy Attack	Is a flaw in smart contracts that allows an attacker to take control of the system by repeatedly entering a function before it finishes. Example: The DAO Attack	Mitigation: Use the checks-effects-interactions pattern to prevent reentrant calls.
2. Arithmetic Overflow/Underflow	Unexpected outcomes can emerge from integer overflow and underflow, which happen when the output of a calculation surpasses the maximum or lowest values that is stored in a variable.	Mitigation: Implement checks to prevent overflow/underflow and use safe mathematical libraries value
3. Denial of Service (DoS)	Is a malicious attempt to overwhelm a server or website with a flood of traffic, rendering it unavailable to its intended users. Example: Gasless Sending	Mitigation: Implement gas limits, avoid complex calculations in loops, and use external adapters for heavy operations.
4. Block timestamp manipulation	It involves deliberately altering the timestamp of a block on a blockchain network, with the aim of taking advantage of any vulnerabilities in smart contracts. Example: Timestamp manipulation	Mitigation: Use block number or block timestamp instead of current timestamp for critical operations.
5. Delegate Calls Attack	Delegate call attacks in Solidity happen when a contract passes control to another contract via a delegate call but fails to verify the called contract's inputs, giving attackers the chance to execute arbitrary code and gain control of the contract.	Mitigation: Implement delegate calls carefully, ensuring robust security checks.
6. Self-destruct Attack	is a Solidity function that enables a contract owner to terminate the contract and transfer any remaining payments to a specified location. Example: Parity Wallet Bug	Mitigation: Limit the usage of self-destruct and implement fail-safe mechanisms.
7. Phishing using tx.origin	In this, an attacker exploits the tx.origin function in Solidity smart contracts to trick users into revealing their private keys or other sensitive information, by sending messages which may appear to come from a legitimate source. Example: Fake contract impersonation	Mitigation: Instead of tx.origin, msg.sender should be used for authentication and authorization.
8. Honeypot	A honeypot is a kind of security system that is designed to trap attackers who try to exploit vulnerabilities in the contract.	FAKE Token Honeypot

Motivation

The motivation for this study is to address the increasing reliance on smart contracts in the block chain industry and the potential risks associated with their use. Smart contracts enable financial activities like payments and auctions on block chain systems, but their openness and interaction with other contracts make them vulnerable to security risks (Cointelegraph, 2023). As a result, there is a pressing need for a comprehensive understanding of smart contract's vulnerabilities, their causes, and preventive methods. This study aims to identify critical software safety issues in smart contracts and assess the effectiveness of existing verification frameworks and preventive methods in minimizing security risks. By addressing these concerns, the study targets to reduce the vulnerability of smart contracts and promote their wider adoption in the block chain industry.

Research Questions

RQ-1) What are the most critical software security issues that affect smart contracts on block chain systems, and how do they put the contracts at risk?

RQ-2) what frameworks are available for verifying and performing static analysis of smart contracts to identify security flaws, and how effective are they in preventing security risks?

RQ-3) how can smart contract developers use best practices and principles to create secure smart contracts that minimize the risk of security vulnerabilities?

Overall contributions of this research article are characterized as follows:

1. Overview of Ethereum protocol (working and history) and Smart Contracts.
2. An overview of 9 common smart contract vulnerabilities and real-world Smart Contract Security Incidents. Vulnerabilities like: reentrancy attack, self-destruct attack, DOS attack, timestamp manipulation etc.)
3. Methodology for Identifying Security Issues and secure coding guidelines.
4. Comparison of various smart contract auditing tools (like securify, Smart Check etc) with each other and with paper proposed algorithmic solution.

Literature Review

Smart contracts have become a critical component of block chain technology, facilitating trustless transactions in a decentralized environment. However, the trustless nature of smart contracts is only possible when security issues are addressed, as vulnerabilities can be exploited by malicious actors. Thus, several studies have been

conducted to identify and address security issues in smart contracts. This literature review summarizes and compares 19 research papers on smart contract security.

Smart contract security is an essential aspect of blockchain technology that has seen significant attention from both academia and industry. Researchers have developed various techniques for identifying security vulnerabilities in smart contracts, including comparing history versions (Chen, 2020), static analysis (Ye *et al.*, 2020), security assessment integrated frameworks (Jain *et al.*, 2022), and semantic-based defect detection (Wang *et al.*, 2022). Practitioners have also provided their perspectives on smart contract security (Wan *et al.*, 2021; Cointelegraph, 2023). Several studies have focused on the security vulnerabilities of Ethereum smart contracts (Luu *et al.*, 2016; Chen, 2020a; Chen *et al.*, 2020b; Hwang and Ryu, 2020; Kushwaha *et al.*, 2022). These studies have identified different types of vulnerabilities, including reentrancy vulnerabilities (Ye *et al.*, 2020), integer bugs (Torres *et al.*, 2018), and authentication issues (Kushwaha *et al.*, 2022). Researchers have also explored the gap between theory and practice in patching vulnerabilities in Solidity, used for developing Ethereum smart contracts (Hwang and Ryu, 2020). Other studies have focused on the safety analysis of smart contracts in general (Polishchuk *et al.*, 2012; Zhou *et al.*, 2018; Liu and Liu, 2019), as well as potential threats of smart contracts in specific blockchain networks such as Hyperledger Fabric (Jain *et al.*, 2022). Finally, several news articles have highlighted the need for smart contract security, such as these three article (Cointelegraph, 2023) describe security exploits and hacks on various token bridges used in the decentralized finance (DeFi) space. (Crypto Times, 2022; Coin Telegraph, 2023) reports on the security exploit of the Nomad Token Bridge, where hackers were able to drain \$190 million in funds. The article mentions that the exploit was related to the manipulation of the bridge's smart contract, and that the team behind the bridge was working to identify the vulnerability and mitigate the damage. (Crypto Times, 2022) describes the compromise of the Wormhole Bridge, which led to a theft of \$321 million ether (ETH). The article notes that the hackers were able to exploit a vulnerability in the bridge's code, and that the attack was one of the largest hacks of 2022. (Bitcoin, 2022) reports on the siphoning of \$80 million from the Qubit Cross-Chain Bridge, which was the largest DeFi exploit of 2022 at the time of the article's publication. The article mentions that the exploit was related to a flaw in the bridge's code, and that the team behind the bridge was working to address

the issue and compensate affected users. In (Chainlink, 2021), the Introduction to Chainlink VRF provides a detailed introduction to Chainlink VRF and its applications. The authors explain the key features of Chainlink VRF and how it generates secure and verifiable random numbers in smart contract applications. Moving on, Open Zeppelin (2019). Math library documentation is a comprehensive guide to the mathematical operations used in smart contract development. It provides a valuable resource for developers who need to perform complex calculations in their smart contract code. In (Buterin, 2014), the Ethereum White Paper outlines the key features of Ethereum as a smart contract and decentralized application platform. The author, Vitalik Buterin, discusses the technical details of Ethereum's architecture and design principles, providing insights into how Ethereum can be used to create a wide range of decentralized applications. Finally, (Atzei *et al.*, 2017) presents the same Ethereum White Paper authored by Gavin Wood, which delves into the technical details of Ethereum's architecture and design principles, providing a comprehensive understanding of the platform's capabilities. (Atzei *et al.*, 2017) provides a comprehensive survey of various attacks on Ethereum-based smart contracts. The authors outline different types of attacks, including reentrancy, denial of service, and timestamp dependence attacks, among others. They also discuss how these attacks can be prevented and provide some examples of secure smart contracts. In (Luu *et al.*, 2016), the authors present a system for making smart contracts more secure by improving their programming language. The system, called Oyente, uses symbolic execution and static analysis to identify vulnerabilities and provides suggestions for improving their security. (Nikolic *et al.*, 2018) proposes a system for identifying vulnerable smart contracts on the Ethereum network. The authors use static analysis to detect three types of vulnerable contracts: greedy, prodigal, and suicidal. The system is capable of analyzing a large number of smart contracts and can be used to improve the security of the Ethereum network. (Wang *et al.*, 2019) proposes a testing suite for smart contracts called ContractLab. The authors argue that smart contracts are difficult to test and that there is a need for a comprehensive testing suite. ContractLab includes various types of tests, including functional, security, and performance tests. (Perez and Livshits, 2021) finds that the majority of smart contracts have at least one vulnerability, but most of them are low severity. Smart contract developers should also consider the likelihood of exploitation when prioritizing which vulnerabilities to address. (Li *et al.*, 2019) discusses

denial of service (DOS) attacks on Ethereum-based smart contracts. The authors analyze the causes of these attacks and propose some defenses against them. Solidity provides documentation for Solidity, a contract-oriented programming language used for writing smart contracts on the Ethereum network. The documentation covers the syntax and semantics of Solidity and provides examples of how to write smart contracts. (Ali *et al.*, 2022) presents a survey of smart contracts that covers their architecture, applications, and future directions. The authors discuss the advantages and challenges of smart contracts and provide examples of their use in various domains. (Antonopoulos *et al.*, 2018) is a book that provides a comprehensive guide to building smart contracts and decentralized applications (DApps) on the Ethereum network. The authors cover various topics, including the basics of blockchain technology, the Ethereum network, and the Solidity programming language. (Habashneh *et al.*, 2020) proposes a system for understanding and mitigating block timestamp manipulation attacks in blockchain-based smart contracts. The authors analyze the causes of these attacks and propose some solutions for preventing them. (Islam *et al.*, 2020) identifies arithmetic overflow and underflow vulnerabilities in Solidity smart contracts. The authors propose a system for detecting and preventing these vulnerabilities. (Sriram *et al.*, 2019) discusses the insecure source of randomness in Solidity smart contracts. The authors propose some solutions for improving the randomness of smart contracts. (Kumar *et al.*, 2018) proposes a system for understanding and mitigating the self-destruct vulnerability in Ethereum smart contracts. The authors analyze the causes of this vulnerability and propose some solutions for preventing it. (Goyal *et al.*, 2020) discusses a new vulnerability in Ethereum smart contracts called phishing with tx.origin. The authors propose some defenses against this vulnerability. (Kamble *et al.*, 2021) proposes a honeypot-based security mechanism for smart contracts on blockchain. The authors argue that honeypots can be used to detect and prevent attacks on smart contracts. (Li *et al.*, 2020) proposes a system for understanding and mitigating delegate call attacks in Solidity smart contracts. The authors analyze the causes of these attacks and propose some solutions for preventing them. (Drake *et al.*, 2021) provides a comprehensive guide to Ethereum 2.0 and the Ethereum merge. The guide covers the basics of Ethereum 2.0, including its design, security, and scalability, and provides some examples of its use in various domains. (Delmolino *et al.*, 2016) conducted a study highlighting the lessons learned from their experiences in a crypto Currency lab and provided

insights into creating secure smart contracts. It also offers valuable guidance for developers and researchers aiming to create secure smart contracts, highlighting important considerations and best practices.

Researchers and developers have been actively exploring techniques and tools to detect and mitigate these vulnerabilities. A method that takes this into account is symbolic execution, which is proposed in (Yao *et al.*, 2022) as a better vulnerability detection mechanism for smart contracts. They hope that by utilising symbolic execution techniques, this approach can improve the accuracy and efficiency of vulnerability discovery. SmartCheck, a static analysis tool for Ethereum smart contracts, is introduced in (Tikhomirov *et al.*, 2018), another method used for smart contract security analysis. Their work focuses on identifying common vulnerabilities and security issues by analyzing the contract's source code. The authors demonstrate the effectiveness of Smart Check in detecting vulnerabilities and preventing potential exploits. In addition to dedicated tools, existing frameworks and platforms can also contribute to smart contract security. ConsenSys' Mythril (Torres *et al.*, 2018) is an example of a widely-used security analysis tool for Ethereum smart contracts. Mythril employs symbolic execution and other techniques to identify vulnerabilities, such as reentrancy and integer bugs. The tool is actively maintained and provides valuable insights for developers seeking to enhance the security of their contracts.

The Osiris framework (Torres *et al.*, 2018) solves the problem of integer issues in smart contracts built on Ethereum. Osiris helps ensure that smart contract implementations are safe from these types of flaws by actively searching for them. When the authors discuss their findings, they aim to highlight how critical it is to track down and fix integer errors. For binaries and smart contracts, Manticore (Mossberg *et al.*, 2019) provides a straight forward symbolic execution environment. The goal of the framework is to make finding and analysing vulnerabilities easier by offering a user-friendly interface. Manticore demonstrates its effectiveness in the field of automated software engineering and showcases its potential impact on smart contract security. Also worth mentioning is Securify (Tsankov *et al.*, 2018), which employs static analysis techniques to do a practical security study of smart contracts, looking for holes in the contract's authentication, authorization, and other security-related features. In (Lu *et al.*, 2021), a more useful tool for analysing the safety of Ethereum smart contracts was introduced: NeuCheck. When it comes to security analysis, NeuCheck places a premium on usability and efficiency. To find

vulnerabilities like as reentrancy and transaction ordering problems, the authors suggest a method that blends static analysis with concrete symbolic execution. With its emphasis on composite vulnerabilities, Ethainter (Brent *et al.*, 2020) is a smart contract security analyzer. Ethainter identifies vulnerabilities that arise due to the interaction of multiple smart contracts, enabling developers to address complex security issues.

The development and evolution of Ethereum have been extensively documented through a range of resources and publications. This literature review examines key references that provide insights into the foundations, concepts, and advancements surrounding Ethereum (Buterin, 2014). represents a landmark work that outlined Ethereum's potential as a platform for decentralised applications and smart contracts of the future. This paper laid the groundwork for the Ethereum ecosystem and outlined its potential applications in various industries. This publication further expanded on the vision and technical aspects of Ethereum, emphasizing its role in enabling decentralized applications and smart contract execution. (Antonopoulos *et al.*, 2018) offers an in-depth exploration of Ethereum's capabilities. This book delves into the practical aspects of building and deploying smart contracts, providing developers and enthusiasts with comprehensive guidance on leveraging the Ethereum platform. (Drake *et al.*, 2021) provide an up-to-date understanding of the Ethereum ecosystem. This comprehensive guide covers the transition to Ethereum 2.0, highlighting the advancements and changes brought about by the Ethereum Merge. In terms of technical specifications, (Wood., 2014) presents a secure, decentralized, and Generalized transaction ledger. This paper provides detailed insights into Ethereum's underlying protocol, addressing aspects such as consensus mechanisms, transaction processing, and security considerations. The original Ethereum white paper by Vitalik Buterin (Buterin, 2014) is an essential reference that outlines the fundamental principles and design philosophy behind Ethereum. It serves as a foundational piece for understanding Ethereum's goals and the vision of a decentralized, trustless, and programmable platform. For a comprehensive review of Ethereum's platform and perspectives (Lubin and Wood, 2014), offers valuable insights. This publication explores Ethereum's architecture, features, and potential use cases, providing a comprehensive overview of the platform's capabilities. In the broader context of blockchain and decentralized technologies, references such as "Blockchain: Blueprint for a New Economy" by Swan (Swan, 2015) and "Bitcoin and Cryptocurrency Technologies: A Comprehensive

Introduction" by Bonneau, Felten, Miller, and Goldfeder (Narayanan *et al.*, 2016) offer valuable insights into the broader landscape and the oretical foundations of decentralized technologies. Lastly, Popov's white paper (Popov, 2016) introduces an alternative distributed ledger technology known as the Tangle. While not directly related to Ethereum, this publication showcases the diversity of approaches within the broader field of decentralized ledger technologies.

As a result of the development of various methodologies, tools, and frameworks targeted at identifying vulnerabilities, reducing risks, and boosting overall security, tremendous progress has been made in the field of smart contract security. Symbolic execution, static analysis, and practical security analysis tools play crucial roles in identifying vulnerabilities and promoting secure coding practices during smart contract development. However, despite these advancements, there remain significant challenges in bridging the gap between theory and practice, as well as the need for continuous security testing and updates. To ensure the robustness of Ethereum-based smart contracts, ongoing research and development efforts are essential in addressing emerging challenges and maintaining the security of smart contract ecosystems.

Ethereum Smart Contract: Working And Structure

A smart contract is a self-operating contract that operates on the Ethereum block chain, as described in the Ethereum white paper. It is a piece of code that runs on the Ethereum Virtual Machine (EVM) and is executed by the network nodes. Whenever a user sends a transaction to a smart contract, the transaction is first broadcasted to the network and then picked up by a miner who adds it to a block (Antonopoulos *et al.*, 2018). Following this, the EVM processes the transaction and executes the smart contract code in a step-by-step manner using a stack-based approach (Buterin, 2014). As each operation in the code is executed sequentially, the results are stored on a stack. Subsequently, the next operation utilizes the result from the stack as input, and this process continues until the code has been completely executed. See Figure 1. When executing a smart contract, the EVM performs several important functions:

Gas Management: Every operation in the smart contract code requires gas to execute. Gas is a measure of computational effort and is used to prevent spamming and denial of service attacks on the Ethereum network. The EVM manages the gas required to execute the smart contract and charges

the sender of the transaction based on the gas consumed.

State Management: The EVM is responsible for maintaining the state of the smart contract, which inculcates the values of its state variables. Whenever a function modifies the state of the contract, the EVM updates the state on the blockchain.

Security Checks: The EVM performs several security checks to make sure that the smart contract code does not pose a security risk to the network. For example, the EVM checks for potential reentrancy attacks, where a malicious contract can Repeatedly call itself to drain funds from other contracts.

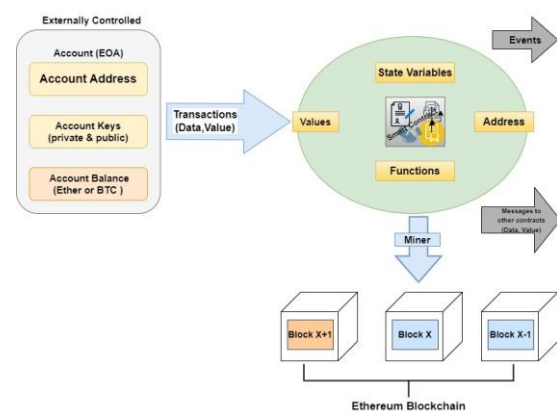


Figure 1. Smart Contracts in Ethereum Block-chain Network

After a smart contract is created, it is deployed onto the Ethereum network and is then given a unique address that identifies it on the network (Antonopoulos *et al.*, 2018). The creator of the contract can specify the rules and conditions of the contract within its code. Once the code is deployed, the contract is immutable and cannot be modified. Smart contracts can both receive and send Ether, which is the indigenous crypto currency of the Ethereum network and can execute certain predefined actions based on certain conditions. An example of this is a smart contract that releases funds from an escrow account once certain pre-specified conditions have been met. The structure of an Ethereum smart contract can be divided into several main components:

Pragma statement: This is the first line of code in a Solidity contract and specifies the version of the Solidity compiler to be used for compiling the contract.

Contract declaration: This is the main body of the contract and is enclosed within curly braces {}. It contains the code that defines the contract's state variables, functions, events, and modifiers.

State variables: These are variables that represent the state of the contract and are stored on the Ethereum blockchain. They can be accessed by any function within the contract.

Functions: These are the methods that define the behavior of the contract. They can be called by external accounts, other contracts, or by the contract itself. Functions can modify the state of the contract and can return values to the calling entity.

Events: These are messages that can be emitted by the contract to notify external accounts or other contracts of important state changes. They can be used for debugging, monitoring, or for triggering external actions.

Modifiers: These are functions that can be used to modify the behavior of other functions within the contract. They are typically used for access control, validation, or other common tasks that need to be performed by multiple functions within the contract.

Constructor: This special function is executed once when the contract is deployed. It is used to initialize the state variables of the contract.

Fallback function: This is a special function that is executed when the contract receives a transaction not matching any of the defined functions within the contract. It can be used to handle unexpected transactions, or to implement custom functionality.

Overall, the structure of an Ethereum smart contract is designed to provide a flexible and extensible framework for implementing complex business logic on the blockchain (Ali *et al.*, 2022). By leveraging the power of Solidity, developers can create smart contracts that can automate a wide range of business processes and enable decentralized, trustless transactions (Ali *et al.*, 2022).

Ethereum Smart Contract: Vulnerability and Prevention

Block Timestamp Manipulation: Block timestamp manipulation refers to the act of intentionally changing the timestamp (see Figure 2) of a block in a blockchain network in order to exploit a vulnerability in smart contracts (Habashneh *et al.*, 2020). In the Ethereum network, each block contains a timestamp that records the time when the block was created. The figures in blockchain (refer to Figure 2) are expressed as Unix time, which indicates the seconds that have passed

since 00:00:00 UTC on January 1, 1970. Unix time is widely used for representing timestamps because it is easy to convert to and from human-readable date and time formats.

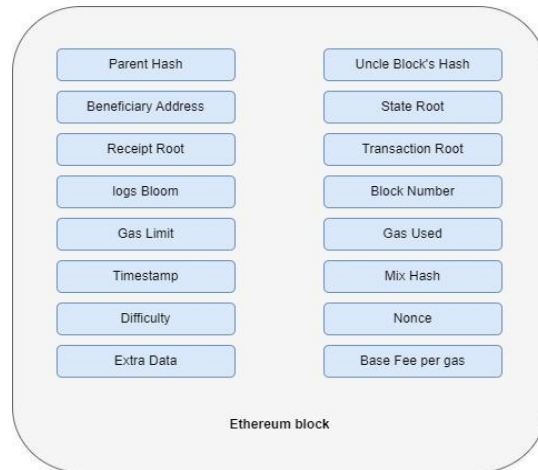


Figure 2. Ethereum Block Architecture

In a smart contract the timestamp value can be accessed with the help of `block.timestamp`. This value is usually used to generate randomness, implement features requiring a timelock and NFT minting. Consider the algorithm 1 below, where a function uses `block.timestamp` (see Figure 2) for inducing randomness to transfer the entire contract balance to the caller (i.e., the address that initiated the function call) using the "call" function. This is considered as a bad practice, because the miners with enough hash power can manipulate timestamps (Habashneh *et al.*, 2020) and can thus attempt to drain all the balance of the smart contract. Even after the Ethereummerge (consensus mechanism changed from Proof of Work to Proof of Stake), it is still considered as a vulnerability (Drake *et al.*, 2021).

Algorithm 1. Algorithm to demonstrate block. Timestamp in Smart Contract

```
function vulnerable Lottery Contract () external payable {
    Modifier to ensure msg.value>1 ether
    // Malicious miner can manipulate timestamps to drain
    // all the lottery contract balance.
    if(block.timestamp%7 ==0){
        Pay msg.sender value: address(this).balance
    }
}
```

Table 2. Comparison of Auditing Tools for Block Timestamp Manipulation

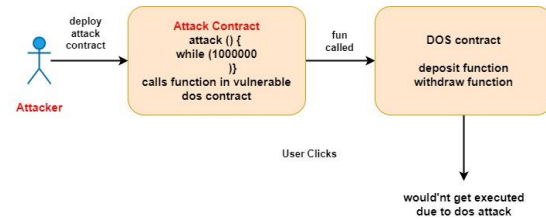
Auditing Tool	Timestamp manipulation Detection/ Prevention	Features
Chainlink VRF	Yes	A safe and impenetrable source of randomness is offered by Chainlink VRF (Verifiable Random Function), which can lessen the effects of timestamp tampering.
SmartCheck	Yes	Analyzes the contract's source code and identifying potential issues related to the usage of block. Timestamp.
Mythril	Yes	Mythril has detection capabilities for timestamp manipulation vulnerabilities.
Manticore	Yes	Manticore is a symbolic execution tool that can explore different execution paths, including timestamp-related calculations
Securify	Yes	Securify utilizes static analysis techniques to identify timestamp manipulation vulnerabilities.
Neucheck	No	Neucheck primarily focuses on analyzing delegate call vulnerabilities and does not explicitly provide timestamp manipulation detection capabilities.
Ethainter	No	Ethainter focuses on analyzing Ethereum smart contracts for vulnerabilities related to business logic and security patterns but does not explicitly target timestamp manipulation vulnerabilities.

Hence using block. Timestamp as a source of randomness is a bad practice. To produce random numbers in smart contracts other safe methods can be used. One such method is Chain link VRF. Chain link VRF works by allowing a user to request to the network for a random number. A set of oracles is then selected based on reputation, performance history, and other factors, and each oracle performs a computation using a seed value and other inputs. The results are aggregated using a consensus algorithm to ensure accuracy, and the result is sent back to the user with a proof of authenticity. This mechanism allows for secure and decentralized random number generation (Table 2).

Denial of Service: In the context of Ethereum smart contracts, DOS (short for Denial of Service) refers to an attack that exploits vulnerabilities in a smart contract's code to overwhelm the network and prevent transactions from being processed or validated (Li. *et al.*, 2019) Occurs when a malicious User exploits a flaw within the contract that allows them to Perform an operation that consumes an excessive number of Resources or computational power. This attack is resource Exhaustion attack. (See Figure 3).

The code in algorithm 2 is vulnerable to a Denial of Service (Resource Exhaustion) attack. The vulnerability is in the Attack contract's function, which repeatedly calls the deposit function in the DOS contract without pause (by sending an amount of ether with each call), causing it to execute 100,000 times and flood the system with requests. This makes the system become overwhelmed and

unresponsive for the users who are further prevented from processing legitimate transactions.

**Figure 3. DOS Attack in Smart Contract**

Algorithm 2. Smart Contract Vulnerable to DOS Attack

```

contract Vulnerable {
    function deposit () {
        require (msg.value> 0);
    }
    //other crucial functions of this contract gets impacted....
}
contract Attack {
    COUNTER=0
    function attack {
        WHILE(COUNTER<1000000) {
            Each iteration deposits 1 Wei in Vulnerable contract
        }
    }
}
  
```

Although, a new defense mechanism called "E-THGuard," which uses an adaptive algorithm to detect and mitigate DoS attacks on smart contracts (Li *et al.*, 2019), to mitigate this vulnerability, the deposit function in the DOS contract should include a limit on the amount of ether that can be deposited per transaction and per address (see Algorithm 3). Additionally, the DOS contract can implement rate limiting or circuit breaking mechanisms to prevent an attacker from overwhelming the system. This is done by including the `require(msg.value>0 && reputation[msg.sender]<=100)` modifier in the deposit function below. This modifier ensures that a particular address doesn't call the deposit function more than 100 times (Table 3).

Algorithm 3. Smart Contract Immune to DOS Attack

```
contract Vulnerable {
    mapping (address => uint256) balances;
    mapping (address => uint256) reputation;
    function deposit () {
        require(msg.value>0&&reputation[msg.sender]<=
            100);
        balances[msg.sender] += msg.value;
        reputation[msg.sender]++;
    }
}
```

Reentrancy: In a reentrancy attack, an attacker exploits a vulnerability in the smart contract

code that allows them to call the same function repeatedly before it has finished executing (Ye *et al.*, 2020; Drake *et al.*, 2021). It allows an attacker to execute malicious code and steal funds before the original function has been completed.

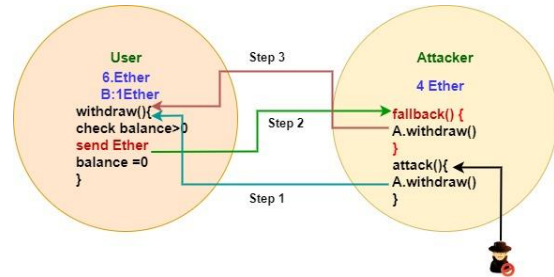


Figure 4. Reentrancy Attack in Smart Contracts

As shown in the Figure 4, the attacker contract executes the attack function, triggering the withdraw function in the user contract. In the withdraw function, after checking the balance of the attacker it sends required ether to the attacker contract, which provokes the fallback function in the attacker contract. The fallback in return calls the withdraw function in the user contract which again deposits ether in the attacker contract. This loop of sending ether is executed until the balance of the user contract is emptied (Drake *et al.*, 2021). See

Table 3. Comparison of Auditing Tools for Denial of Service

Auditing Tool	Dos Detection/Prevention	Features
Algorithm -3	Yes	This contract is immune to a DoS attack because the deposit function has a reputation-based restriction that prevents users with a reputation higher than 100 from depositing more than 1 ether, limiting the amount of funds that can be deposited by any individual.
SmartCheck	Partial	SmartCheck can identify potential vulnerabilities that could lead to DoS attacks, such as gas inefficiencies, loop conditions, or improper resource allocation. However, it may not cover all possible DoS attack vectors.
Mythril	Partial	Mythril includes detection capabilities for some DoS attack vectors, such as gas limit vulnerabilities, but it may not cover all possible DoS scenarios.
Manticore	Yes	Manticore, with its symbolic execution capabilities, can explore different execution paths and detect potential vulnerabilities, including those related to DoS attacks. It can identify gas-related vulnerabilities and resource exhaustion issues.
Securify	Partial	Securify can detect certain types of DoS vulnerabilities, such as improper gas usage or potential infinite loops. However, it may not cover all possible DoS attack vectors.
Neucheck	Yes	Neucheck specifically analyzes smart contracts for potential DoS attack vectors. It aims to identify vulnerabilities that could lead to DoS attacks
Ethainter	Partial	Ethainter focuses on analyzing smart contracts for security patterns and business logic vulnerabilities. While it may detect certain DoS-related issues, it may not provide comprehensive coverage for all possible DoS attack vectors.

Algorithm 4. Smart Contract Vulnerable to Reentrancy Attack

```

contract Vulnerable {
    mapping(address => uint) balancesMap;
    function deposit() {
        balancesMap[msg.sender] += msg.value;
    }
    function withdraw() public {
        Send value:
        balancesMap[msg.sender] to
        msg.sender.call
        balancesMap[msg.sender] = 0;
    }
}
contract Attack {
    function attack() {
        Deposit 1 ether in vulnerable contract;
        Call withdraw() function in
        vulnerable contract;
    }
    receive() external payable {
        Call withdraw() function in
        vulnerable contract;
    }
}

```

To make a contract immune to re-entrancy attacks using the mutex method, we need to use a Boolean variable “lock”, a mutex (short for mutual exclusion) to enforce a lock on the critical section of code where the state of contract is modified. In algorithm 5, the withdraw function is included with a modifier, which before executing the critical section of code checks whether the lock variable is set to false. If it is not, we revert the transaction with an error message indicating that the function is locked. Once we have verified that the function is not locked, we set the lock variable to true. This prevents any other transactions from executing the critical section of code until the current transaction is complete.

Algorithm 5. Smart Contract Immune to Reentrancy Attack

```

contract Vulnerable {
    mapping(address => uint) balances;
    bool locked = false;
    function deposit() {
        balances[msg.sender] += msg.value;
    }
    function withdraw() public {
        require(locked==false, "Locked");
        locked= true;
    }
}

```

```

Send value =
balances[msg.sender] to
msg.sender.call
balances[msg.sender] = 0;
locked = false;
}

```

Honeypot: Honeypots in smart contracts work by creating a trap that is hidden within the code of the contract. This trap is designed to look like a vulnerability to an attacker. However, when the attacker tries to exploit the vulnerability, they are instead directed to a trap that prevents them from carrying out their attack (Reentrancy attack) (Kamble *et al.*, 2021).

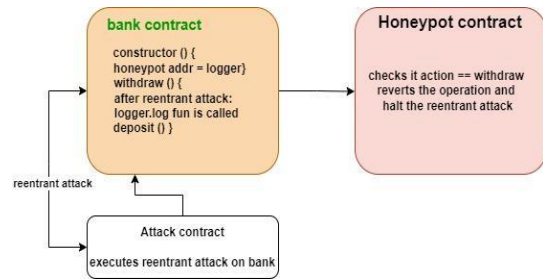


Figure 5. Honey pot Implementation In a Smart Contract

Algorithm 6. Honeypot Implementation in Smart Contract

```

contract Bank {
    Logger logger;
    constructor (Logger _logger) {
        // _logger is address at which Honeypot
        contract is // deployed
        logger = Logger(_logger);
    }

    function withdraw(uint amount) {
        Transfers “amount” to msg.sender
        // after the reentrancy attack is finished the
        code
        // below will be executed. Without this
        execution
        // transfer of balance won't succeed.

        // Honeypot contract log's function is called.
        logger.log(msg.sender, amount, "Withdraw");
    }
}

```

```

function deposit() {
    Deposits msg.value amount in Bank Contract
}

// will be deployed to ethers can to deceive the
attacker

contract Logger {
    event Log(address caller, uint amount, string
action);
    function log(address _caller, uint _amount,
string memory _action) public {
        emit Log(_caller, _amount, _action);
    }
}

contract Honeypot {
function log(address _caller, uint _amount, string
memory _action) public {
    if (equal(_action, "Withdraw")) {

        // transaction is reverted & attacker is
unable to
        //withdraw his funds.
        // attacker's address is revealed on etherscan
        revert("It's a trap");
    }
}

contract Attack {
    function attack() public payable {
        // hacker trying to execute reentrancy attack
        Deposits 1 ether in Bank contract
        Withdraw 1 ether from Bank Contract
    }
    receive() external payable {
        Withdraw 1 ether from Bank Contract if
(address(bank).balance>= 1 ether)
    }
}

```

OUTPUT (Hardhat Testing Environment)

Contract: Bank

- should deposit funds successfully (123ms)
- should withdraw funds successfully (123ms)

Contract: Honeypot

- should prevent reentrancy attack (123ms)
- Bank Contract Address: 0xAbCdEf0123456789
Bank Contract Balance: 5 ether
- Attacker's Address: 0x1234567890AbCdEf
Balance before attack: 1 ether
Balance after attack: 0 ether

- Execution Results:

- Bank Contract - deposit function:
- Execution time: 2 seconds
- Successful

- Bank Contract - withdraw function:
- Execution time: 1 second
- Successful

- Honeypot Contract - log function:
- Execution time: 1 second
- Log message: "It's a trap"

In algorithm 6, the Solidity code implements a honeypot by tricking attackers into thinking they can exploit the Bank contract using a reentrancy (Ye *et al.*, 2020) attack (Kamble *et al.*, 2021). The Bank contract logs all transactions made to it using a Logger contract instance. However, the Logger contract is a honeypot that only logs transactions but doesn't perform any useful functionality. When an attacker tries to withdraw funds from the Bank contract using a reentrancy attack, the withdraw () function first logs the transaction using the Logger contract. It then deducts the amount being withdrawn from the sender's balance, sends the requested amount to the attacker's account, and prevents reentrancy attacks from stealing any additional funds. (See Figure: 5)

However, the honeypot is triggered when the attacker tries to withdraw funds using a reentrancy attack (Kamble *et al.*, 2021). If the transaction is a "Withdraw" transaction, the honeypot reverts the transaction and prints "It's a trap" to the console, preventing the attacker from stealing any funds. The Attack contract is used to test the honeypot by depositing 1 ether into the Bank contract and trying to withdraw 1 ether using a reentrancy attack. The honeypot reverts the transaction, and the attacker's funds remain safe in the Bank contract. The receive () function in the Attack contract is used to execute the reentrancy attack by repeatedly calling the withdraw() function on the Bank contract.

Self - Destruct: In Solidity, there is a function called self-destruct which allows the owner of a contract to destroy it and send remaining funds in the contract to a designated address (see Figure 6). However, if the self-destruct function is not properly implemented, it can create a vulnerability (Kumar *et al.*, 2018) that allows attackers to induce Denial of Service attack in the contract.

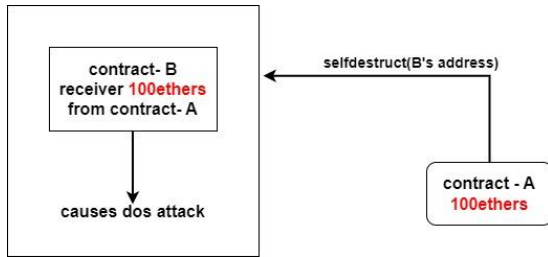


Figure 6. Self Destruct In Smart Contracts

According to the provided figure the contract B is vulnerable to a denial-of-service (DoS) attack due to the use of the self destruct method. The code consists of two contracts: B and A. The B contract has a send Ether function for minting tokens (hypothetically). The contract A is used to perform the DoS attack. In the B contract, there is a goal variable representing the target balance (100 ether) that needs to be reached to execute send Ether function. When the send Ether function is called, it requires the sender to send exactly 1 ether. The function then increases the balance by the amount sent, checks if the balance exceeds the goal, and if not, mints a token by incrementing total Supply and assigns the token to the msg.sender. In the attack function, the address of the B contract is obtained and the self destruct function is called with B contract address as the argument. The self destruct function destroys the contract A and sends its remaining Ether balance to the specified address (B's Address). By calling self destruct (B's Address), all the Ether balance stored in contract-A is transferred to the B contract. However, the B contract does not handle incoming Ether transfers in any meaningful way. This means that if the Attack contract's goal (100 ether), the self destruct call will prevent further execution of the mint function in the B contract. This effectively locks the B contract, preventing any further token minting (rendering the B contract useless).

Consider a contract "B" (see Algorithm 7) which accepts transactions of 1 ether (max) only till its balance is ≤ 100 ether. If a contract "A" (with balance of 100 ethers) executes self destruct function with the address of contract B as a parameter, the contract A gets destroyed from the Ethereum blockchain and all of its balance gets transferred to contract B (figure 6). And now the address(this). The balance of contract B becomes greater than 100 ether, due to which it won't allow other externally owned accounts to deposit any ether in it. Thus, causing a denial of service in contract B.

Algorithm 7. Smart Contract Vulnerable to Self-Destruct Attack

```

contract B{
    uint public goal = 100 ether;
    function sendEther() {
        require(address(this).balance<=
            goal, "Minting is finished!");
        .. Other functions to follow ...
    }
}

contract A{
    // Holds 100 ether
    function attack() public payable {
        selfdestruct(address of contract B);
    }
}
  
```

To resolve this issue, the vulnerable contract B, can be modified by (algorithm 8) using a balance variable instead of address(this).balance to check the overall balance in contract B. This balance variable can be checked against the threshold of 100 ether and incremented with msg.value, each time the send Ether function is called to initiate a transaction by an externally owned account (Table 4).

Algorithm 8. Smart Contract Immune to Self Destruct Attack

```

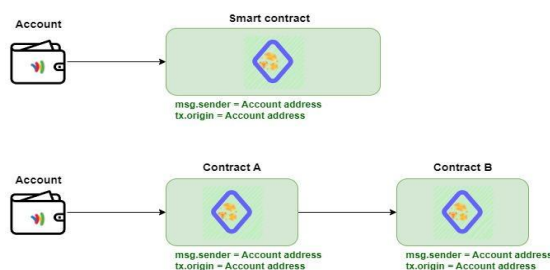
contract B{
    uint public goal = 100 ether;
    uint public balance;
    function sendEther() {
        require(balance <= goal, "Minting is
            finished!");
        balance += msg.value;
        .. Other functions to follow ...
    }
}
  
```

Phishing with tx.origin: in this type of attack, the attacker exploits the tx.origin function in smart contracts to trick users into revealing their private keys or other sensitive information. The 'tx.origin' function returns the user's address who initiated the transaction. This can be useful for authentication purposes. However, attackers can cause a phishing attack by deploying a malicious contract that checks the tx.origin and sends a message to the user requesting them to provide sensitive information such as private keys. The message may appear to come from a legitimate source or a trusted application.

Table 4. Comparison of Auditing Tools for Self-Destruct

Auditing Tool	Self-DestructDetection/ Prevention	Features
Algorithm 8	Yes	It uses an unsigned integer variable “balance” to keep track of contract’s current balance instead of address(this).balance. Even if selfdestruct is called from attack contract it doesn’t increment balance’s alue.
SmartCheck	Yes	SmartCheck can detect self-destruct attack vulnerabilities by analyzing the contract’s source code
Mythril	Yes	Mythril has detection pabilities for self-destruct ttack vulnerabilities. It uses symbolic execution and static analysis
Manticore	Yes	Manticore is a symbolic execution tool that can explore different execution paths, including self-destruct calls
Securify	Yes	Securify has detection capabilities for self-destruct attacks. It utilizes static analysis and formal verification techniques to identify potential vulnerabilities.
Neuchack	No	Neuchack primarily focuses on analyzing delegate calls and does not explicitly provide detection capabilities for self-destruct attacks.
Ethainter	No	Focuses on analyzing Ethereum smart contracts for vulnerabilities related to business logic and security patterns.

In Solidity, `msg.sender` and `tx.origin` are two special variables that are commonly used to identify the sender of a transaction. However, they have different meanings and should be used in different contexts. ‘`msg.sender`’ refers to the immediate caller of a function, which could be a contract or an external account. In other words, it identifies the address of the entity that initiated the current function call. This variable is often used to implement access control mechanisms, where only certain addresses are allowed to perform specific actions within a contract. ‘`tx.origin`’, on the other hand, refers to the original sender of the transaction that initiated the entire chain of function calls. (See Figure 7) This means that even if a contract is called by another contract, `tx.origin` will still return the external account’s address which initiated the transaction. This variable should be used with caution, as it can be manipulated by attackers to bypass security checks implemented by contracts.

**Figure 7. Working of tx.origin in Smart Contracts****Algorithm 9. Smart Contract Used for Phishing**

```

contract Phisable {
    address public owner;
    constructor(address __owner) {owner = __owner;}
    function sendEther() {
        // user was sending eth to this contract through
        this
        // function
    }
    function withdrawAllBalance (address payable
    recipientAddr) {
        require(tx.origin == owner);
        recipientAddr.transfer(address(this).balance);
    }
}

contract Attack {
    // attacker has to ensure that user clicks on this
    function
    function UnsafeOperation() payable {
        phisableContract.withdrawAllBalance(attacke
        r);
    }
}

```

The vulnerability known as a “`tx.origin`” attack, is demonstrated in algorithm 9. The attacker can call the “UnsafeOperation” function by tricking the user into believing that this function can be trusted. This is ensured because the “withdrawAll” function checks whether the “`tx.origin`” (i.e., the original sender of the transaction) is the same as the “owner” of the contract (which establishes trust in the user), then the function proceeds to transfer all

the contract's funds to the specified recipient. The "withdraw All" function in the "Phisable" contract passes their own address as the "_recipient" argument. To perform the attack, the attacker could create a fake interface that looks like the Phisable contract and trick the user into interacting with it instead of the real contract. The attacker could then execute the Unsafe Operation () function, which would call the withdraw All () function with the attacker's address as the recipient, which will cause the user's funds to be transferred to the attacker's address. (See Figure 8) To prevent phishing attacks, it is recommended to use msg.sender instead of tx.origin for authentication purposes. msg.sender returns the address of the immediate sender of the transaction, which is the contract that initiated the call to the current contract. Apart from this, a proposed mechanism works by analyzing the origin and history of a transaction and assigning a score to its authenticity based on various factors such as transaction source, contract source, and code verification (Goyal *et al.*, 2020) (Table 5).

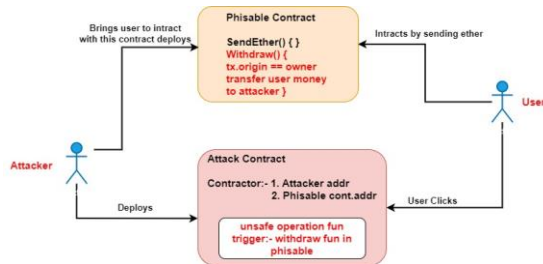


Figure 8. Phishing in Smart Contracts

Arithmetic Overflow & Underflow: Unexpected outcomes can emerge from integer overflow and underflow, which happen when the output of a calculation exceeds the highest or lowest values that is stored in a variable (Torres *et al.*, 2018). (see Figure 9 and Figure 10).

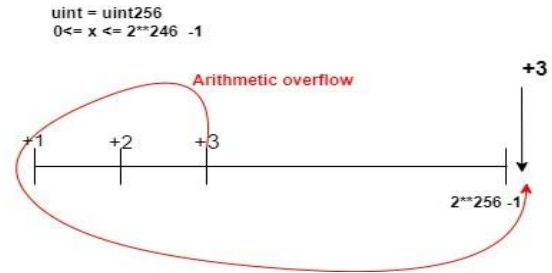


Figure 9. Arithmetic Overflow in Smart Contracts

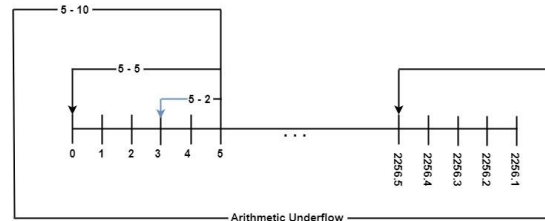


Figure 10. Arithmetic Underflow in Smart Contracts

Table 5. Comparison of Auditing Tools for Phishing

Auditing Tool	Phishing Detection	Features
Using msg.sender	Yes	msg.sender verifies immediate caller, preventing phishing; As tx.origin can be manipulated.
SmartCheck	Yes	It analyses the contract's source code and identifies potential issues related to the misuse of tx.origin for authentication.
Mythril	Yes	It uses symbolic execution and static analysis techniques to identify potential security issues, including misuse of tx.origin.
Manticore	Yes	Manticore is a symbolic execution tool that can explore different execution paths, including those involving tx.origin, to identify vulnerabilities and generate detailed reports.
Security	Yes	It employs static analysis techniques and can identify potential security vulnerabilities, including misuse of tx.origin for authentication purposes.
Neuchek	Yes	It focuses specifically on vulnerabilities that exploit tx.origin for unauthorized access.
Ethainter	No	It does not explicitly target phishing attacks using tx.origin.

Algorithm 10. Smart Contract Immune to Arithmetic Underflow/Overflow

```

contract TimeLock {
    function deposit () {
        Increments balance of msg.sender
        by msg.value
        Increments the lockTime of
        msg.sender by
        int256(block.timestamp + 1
        weeks)
    }
    function increaseTime (int256
    _secondsIncrease) {
        msg.sender'slockTime
        increments by _secondsIncrease;
    }
}
contract Attack {
    function attack() {
        Deposits msg.value in TimeLock
contract
        int256 x = locktime of Attack
        contract in TimeLock Contract
        x *= -1;
        Increase LockTime by x in
        TimeLockcontract
        Withdraw balance from
        TimeLockcontract
    }
}

```

As per algorithm 10, in the Time Lock contract, the function increase Lock Time is vulnerable to arithmetic overflow/underflow. In the original implementation, the increase Lock Time function uses the += operator to increase the value of lock Time of msg. sender by _secondsToIncrease. If _secondsToIncrease is a large enough negative value, this operation can

result in an integer overflow (or vice-versa), causing lock Time [msg.sender] to become a very large positive value or even wrap around to negative values. The vulnerability can be exploited in the Attack contract by depositing some funds into the Time Lock contract and then passing a negative value for _seconds To Increase to the increase Lock Time function (Islam *et al.*, 2020). The function will then add this negative value to lock Time [msg. sender], potentially causing an integer underflow, which reduces the lock time to 0 seconds, due to which, the Attack contract withdraws the funds by contacting the withdraw functionality in the Time Lock contract.

To prevent this, the Open Zeppelin Safe Math library should be used to perform safe arithmetic operations. In the corrected implementation, the increase Lock Time function uses the Safe Math. add function to increase lock Time [msg.sender] by _seconds To Increase. If _seconds To Increase is a large enough negative value, Safe Math. add will throw an exception instead of allowing an integer overflow to occur (Table 6).

Insecure Source of Randomness: is a security weakness that can occur when a system uses a predictable or easily guessable source of variability. Attackers can anticipate the random value and then use it to compromise the security of the system (Sriram *et al.*, 2019). Consider two contracts, "Guess Random Number" and "Attack" The "Guess Random Number" contract allows users to submit a guess of a random number, and if their guess matches the generated number, they will receive the contract's balance. The "Attack" contract aims to exploit the insecure randomness of the "Guess Random Number" contract. See Algorithm 11. The insecure randomness issues arises from the fact that both contracts generate the random number

Table 6. Comparison of Auditing Tools for Arithmetic Over flow & Under flow

Auditing Tool/Library	Arithmetic Underflow/Overflow Detection	Features
Open Zeppelin	Yes	It provides robust and audited contract implementation, which help prevent arithmetic overflow and underflow vulnerabilities.
SmartCheck	Yes	SmartCheck can detect arithmetic overflow and underflow vulnerabilities.
Mythril	Yes	Being a symbolic execution tool, can explore different execution paths, including arithmetic operations, to identify vulnerabilities
Manticore	Yes	Being a symbolic execution tool, can explore different execution paths, including arithmetic operations
Security	Yes	Uses static analysis to identify security vulnerabilities, including arithmetic overflow and underflow.
Neucheck	Yes	It leverages static analysis techniques to identify potential issues related to arithmetic operations
Ethainter	No	It does not explicitly target arithmetic overflow and underflow

using block hash and block.timestamp (Sriram *et al.*, 2019). Since both contracts are executed in the same block, the values returned by these functions will be the same in both contracts. See Figure 11. A solution called "smart seed," which uses a combination of user input and a hash of a secret to generate a random number that is unpredictable and secure (Sriram *et al.*, 2019) (Table 7).

Algorithm 11. Insecure Randomness in Smart Contract

```
contract GuessRandomNumber :
uint balance = 100 ether;
function guess(_guess) {
    ans = keccak256(abi.encodePacked
    (blockhash (block.number - 1),
    block.timestamp) );

    if (_guess == ans): SEND value = balance
    to Attack Contract
}

contract Attack :
function attack(guessRandomNumber) :
    ans =keccak256 (abi.encodePacked
    (blockhash(block.number - 1),
    block.timestamp) );
    guessRandomNumber.guess(ans);
```

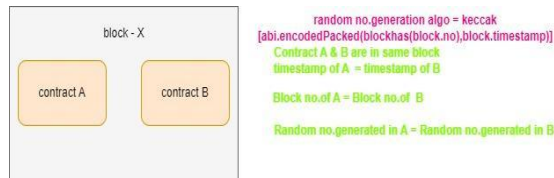


Figure 11. Random No. Generation in Smart Contracts

Delegate Calls: A delegate call attack is a type of attack in Solidity that occurs when a contract passes control to another contract using a delegate call but does not properly validate the called contract's inputs. This can allow an attacker to execute arbitrary code and potentially take control of the contract (Li *et al.*, 2020).

The vulnerabilities due to delegate calls are a result of two key features of Delegate Call. These features are:

1. The context-preserving nature of Delegate Call. (see Figure 12)
2. Same storage layout of both the Caller and Receiver.

Consider three contracts: A, B, and C (See algorithm 12), Contract A has a simple function set Owner that sets the owner variable to the msg.sender address. Contract B has a similar storage layout as contract A. It also has a fallback function that calls the set Owner function of contract A using delegate call. Finally, contract C has an instance of contract B and a function attack that calls the set Owner function of contract B using call and abi.encode With Signature.

The problem with this code is that contract B's fallback function uses 'delegate call' instead of 'call'. Delegate call allows the called function (in this case, set Owner of contract A) to access the storage of the calling contract (in this case, contract B). This means that if contract C calls contract B's fallback function, it can execute the set Owner function of contract A and modify the owner variable of contract B. (See Figure 12) (Table 8).

Table 7. Comparison of Auditing Tools for Insecure Source of Randomness

Auditing Tool	Insecure Randomness Detection	Features
Smart-Seed	Yes	It employs advanced analysis techniques to detect weak or predictable random number generation algorithms
SmartCheck	Yes	SmartCheck can detect insecure randomness vulnerabilities.
Mythril	Yes	It has detection capabilities for insecure randomness. It uses symbolic execution and static analysis
Manticore	Partial	Its focus is more on exploring execution paths and detailed reports on rather than specifically targeting insecure randomness.
Security	Partial	Primarily focuses on static analysis techniques and may not have specific check for insecure randomness.
Neucheck	Yes	It can detect and highlight Potential issues that may arise from weak or predictable random number generation.
Ethainter	Partial	Focuses on analyzing smart contracts for vulnerabilities related to business logic and security patterns.

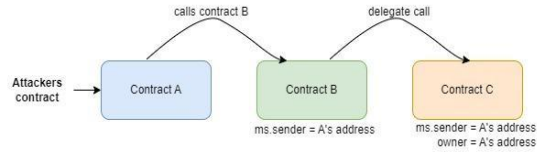


Figure 12. Delegate Call in Smart Contract

Algorithm 12. Delegate Call in Smart Contract

```

contract A1 {
    address public owner;
    function setOwner() : owner = msg.sender;
}
contract B1 {
    address public owner;
    constructor(A1 _a) : owner = msg.sender;
    fallback() : address(a).delegatecall(msg.data);
}
contract C1 {
    function attack():
    b.call(abi.encodeWithSignature("setOwner()"));
}
  
```

Table 8. Comparison of Auditing Tools for Delegate Call

Auditing Tool	Delegate Call Detection	Features
Smart Check	Yes	SmartCheck can detect delegate call vulnerabilities.
Mythril	Yes	It has detection capabilities for delegate call. It uses symbolic execution and static analysis
Manticore	Yes	It is a symbolic execution tool that can explore different execution paths, including delegate calls
Securify	No	Does not explicitly provide delegate call detection capabilities.
Neucheck	Yes	Neucheck is designed to analyze the usage of delegate calls and can detect potential issues related to their usage.

The vulnerability in this code arises from the use of delegate call in contract B's fallback function, which allows an attacker to modify the storage of contract B by calling a function in contract A. This can be exploited by a malicious contract, such as contract C, to modify the state of contract B in unintended ways. The correct way to call a function in a different contract is to use 'call', which does not give the called contract access to the caller's storage.

Results

We conducted a thorough analysis of 9 different types of vulnerabilities in smart contracts, employing various algorithmic methods and exploring the effectiveness of different auditing techniques. By detecting these vulnerabilities early in the development process, we were able to prevent potential attacks on the contract and safeguard user funds. Our auditing process included an assessment of the code against industry best practices, which highlighted areas where improvements could be made to better align with the standards. Although our auditing process required additional time and resources, it ultimately resulted in a more secure and reliable smart contract, and helped avoid costly and time-consuming security incidents in the future.

Based on our findings, we recommended simplifying the code where possible, adding additional input validation checks, and

implementing access controls to prevent unauthorized access to sensitive functions. Interestingly, we observed that automated auditing tools were effective in identifying vulnerabilities such as integer overflow/underflow, while manual auditing was more effective in identifying race conditions, including denial of service and insecure randomness. It is clear that developers need to be mindful of prevalent vulnerabilities and take measures to mitigate them throughout the development process. Furthermore, auditors should consider incorporating both manual and automated techniques for a comprehensive audit, as this can help identify vulnerabilities that may be overlooked by a single method.

Limitations

Our discussion primarily focuses on the security vulnerabilities of smart contracts based on the Ethereum blockchain. While most of the research literature is centered around Ethereum, it is important to note that other blockchain platforms such as EOS, Solana and Hyperledger Fabric also have significant security concerns. However, we have not covered the security vulnerabilities related to these blockchain platforms in this paper and leave this as a topic for future research.

Conclusion

This paper presents a thorough examination of the security vulnerabilities that exist within Ethereum smart contracts. We have identified various types of

security attacks and proposed preventive measures, which we have categorized by their root cause and sub-causes. Moreover, we have given a comprehensive overview of the detection and analysis tools for these attacks, which include different parameters such as tool type, input, analysis type, implementation language, and availability. Our research has uncovered that the security flaws in Ethereum smart contracts make them highly vulnerable to malicious attacks, with potential losses in the millions due to a single vulnerability or error. Therefore, developers of smart contracts should consider these vulnerabilities and adhere to best practices to ensure the creation of secure smart contracts. Lastly, we have provided insights into the current security challenges and future research directions for the development of advanced and robust vulnerability detection tools.

Future Work

The Ethereum blockchain smart contracts are likely to have more features and functionalities added in the future, which may increase the risk of security flaws. Therefore, it is important to improve existing vulnerability detection tools and develop new ones for detecting and mitigating new security threats. Future research could focus on developing platforms that are inherently more secure and resistant to attacks. In (Luu *et al.*, 2016; Zhou *et al.*, 2018; Yamashita *et al.*, 2019; Wan *et al.*, 2021) highlight the risks associated with current smart contract platforms and the need for more secure alternatives. Additionally, efforts should be made to identify unknown vulnerabilities and prevent them, for instance, by incorporating a self-protection (revocable) mechanism in the smart contract to ensure safety in the event of an attack. This is particularly crucial given that smart contracts become immutable once deployed on the blockchain. As Ethereum projects grow larger, scalability and new security issues may arise, and it is therefore essential to address these challenges soon. There is a need for research into real-world attacks and vulnerabilities in smart contracts. Future research could focus on analyzing actual attacks and developing mitigation strategies to prevent similar attacks in the future (Hwang and Ryu, 2020; Kushwaha *et al.*, 2022; Coin telegraph, 2023).

References

- Ali M.H. and Ahmed M.S. (2022). A survey of smart contract: Architecture, applications, and future directions. *Journal of Network and Computer Applications*, 172:1-23

- Antonopoulos, A.M., and Wood, G. (2018). *Mastering ethereum: building smart contracts and dapps*. O'reilly Media.
- Atzei, N., Bartoletti, M., and Cimoli, T. (2017). A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden*, p. 164-186.
- Bitcoin (2022). Hacker Siphons \$80 Million from Qubit Cross-Chain Bridge, Largest DeFi Exploit of 2022 to Date. Retrieved Mar 10, 2023.
- Brent, L., Grech, N., Lagouvardos, S., Scholz, B., and Smaragdakis, Y. (2020). Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 454-469.
- Buterin, V. (2014). A next-generation smart contract and decentralized application platform. white paper, 3(37):2-1.
- Chainlink. (2021). Introduction to Chainlink VRF. Chainlink Documentation. Mar 16, 2023.
- Chen, H., Pendleton, M., Njilla, L., and Xu, S. (2020b). A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)*, 53(3):1-43.
- Chen, J. (2020a, December). Finding ethereum smart contracts security issues by comparing history versions. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 1,382-1,384.
- Cointelegraph (2023). Nomad Token Bridge Drained \$190M in Funds in Security Exploit. Retrieved Mar 10, 2023.
- Crypto Times (2022). Hackers Compromised Wormhole Bridge with Largest Hacks of 2022, Run Away with \$321M Worth WETH. Retrieved Mar 10, 2023.
- Delmolino K., Arnett M., Kosba A., Miller A. and Shi E. (2016). Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab, in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 98-113.
- Drake, J., Ryan, D., and Buterin, V. (2021). *Ethereum 2.0: A Complete Guide to Eth2 and the Ethereum Merge*. Ethereum Foundation. [Online]. Available: <https://ethereum.org/en/eth2/>.
- Goyal A. and Gupta N. (2020). Phishing with tx.origin: A New Vulnerability in Ethereum Smart Contracts. 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT), 1-6.
- Habashneh M.A.Al, Qutayri A.Al, Ghaleb A. and Alkhalidi I.A. (2020). Towards Understanding and Mitigating Block Timestamp Manipulation Attacks in Blockchain-based Smart Contracts. 11th International Conference on Information and Communication Systems (ICICS), 22-27.
- Hwang, S., and Ryu, S. (2020). Gap between theory and practice: An empirical study of security patches in solidity. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 542-553.
- Islam M.R, Hossain M.A, Alam M, Islam M.S. and Ahmed S.U. (2020). Arithmetic Overflow and Underflow Vulnerabilities in Solidity Smart Contracts. 6th International Conference on Advances in Electrical Engineering (ICAEE), 157-162.
- Li, H. Zhang, Y. Xie, and X. Chen (2019). DOS attacks on Ethereum smart contracts: Analyzing the causes and defenses. in *IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 1,274-1,279.
- Jain, S., Shah, N.K., Kurariya, P., Vohra, N., Nandukar, S., Harne, N., and Indraveni, C. (2022). Smart contract-security assessment integrated framework (SC-SIF) for hyperledger fabric. In *2022 IEEE 7th International conference for Convergence in Technology (I2CT)*, 1-11.

- Yamashita K., Nomura Y., Zhou E., Pi B. and Jun S. (2019). Potential Risks of Hyperledger Fabric Smart Contracts. 2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE), 2019, 1-10 Doi: <https://doi.org/10.1109/IWBOSE.2019.8666486>.
- Kamble S., Patil A., and Kadam S. (2021). Honeypot Based Security Mechanisms for Smart Contracts on Blockchain. International Conference on Intelligent Sustainable Systems (ICISS), 832-836.
- Kumar A., Kumar K. and Singh S. (2018). Understanding and Mitigating Self-Destruct Vulnerability in Ethereum Smart Contracts, 3rd International Conference on Computational Systems and Information Technology for Sustainable Solutions (CSITSS), 73-78.
- Kushwaha, S.S., Joshi, S., Singh, D., Kaur, M., and Lee, H.N. (2022). Systematic review of security vulnerabilities in ethereum blockchain smart contract. *IEEE Access*, 10:6,605-6,621.
- Li J., Yu J. and Zhang Z (2020). Understanding and Mitigating Delegate Call Attacks in Solidity Smart Contracts. 2nd International Conference on Computer Science and Software Engineering (CSSE), 61-65.
- Liu, J., and Liu, Z. (2019). A survey on security verification of blockchain smart contracts. *IEEE Access*, 7:77894-77904.
- Lu, N., Wang, B., Zhang, Y., Shi, W., and Esposito, C. (2021). NeuCheck: A more practical Ethereum smart contract security analysis tool. *Software: Practice and Experience*, 51(10):2,065-2,084.
- Lubin, J., and Wood, G. (2014). Ethereum: Platform Review and Perspectives. ETHDEV. Retrieved from <https://ethereum.org/pdfs/EthereumPlatformReview.pdf>
- Luu, L., Chu, D.H., Olickel, H., Saxena, P., and Hobor, A. (2016). Making smart contracts smarter. In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, 254-269.
- Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., and Dinaburg, A. (2019). Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 1,186-1,189.
- Narayanan, A., Bonneau, J., Felten, E., Miller, A., and Goldfeder, S. (2016). Bitcoin and crypto Currency technologies: a comprehensive introduction. Princeton University Press.
- Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A. (2018). Finding the greedy, prodigal, and suicidal contracts at scale. In Proceedings of the 34th annual computer security applications conference, 653-663.
- Open Zeppelin. (2019). Math library. Open Zeppelin Contracts 2.x Documentation. <https://docs.openzeppelin.com/contracts/2.x/api/math>
- Perez, D., and Livshits, B. (2021). Smart contract vulnerabilities: Vulnerable does not imply exploited. In 30th USENIX Security Symposium (USENIX Security 21), 1,325-1,341.
- Poleshchuk, E.M., Shcherbinina, I.A., and Putilova, S.E. (2012). Security Analysis of Smart Contracts in Blockchain Networks. In 2022 Ural-Siberian Conference on Biomedical Engineering, Radioelectronics and Information Technology (USBEREIT), 252-254.
- Popov, S. (2016). The Tangle: A Directed Acyclic Graph (DAG) for Distributed Ledger Technologies. IOTA Foundation. Doi: https://iota.org/IOTA_Whitepaper.pdf
- Sriram K and Sridharan S. (2019). Insecure Source of Randomness in Solidity Smart Contracts. 3rd International Conference on Inventive Systems and Control (ICISC), 1,359-1,362.
- Swan, M. (2015). Blockchain: Blueprint for a new economy. O'Reilly Media, Inc.
- Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., and Alexandrov, Y. (2018). Smartcheck: Static analysis of ethereum smart contracts. In Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain, 9-16.
- Torres, C.F., Schütte, J., and State, R. (2018). Osiris: Hunting for integer bugs in ethereum smart contracts. In Proceedings of the 34th annual computer security applications conference, 664-676.
- Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., and Vechev, M. (2018). Securify: Practical security analysis of smart contracts. In Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, 67-82.
- Wan, Z., Xia, X., Lo, D., Chen, J., Luo, X., and Yang, X. (2021). Smart contract security: A practitioners perspective. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 1,410-1,422.
- Wang, Q., Chen, Z., Liang, X., and Ma, J. (2019). Contract lab: A comprehensive testing suite for smart contracts. Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 370-373.
- Wang, X., Li, J., and Zhang, X. (2022). A semantic-based smart contract defect detection general platform. In 2022 IEEE International Conference on Advances in Electrical Engineering and Computer Applications (AEECA), 34-37.
- Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper, 151(2014):1-32.
- Yao, Y., Li, H., Yang, X., and Le, Y. (2022). An Improved Vulnerability Detection System of Smart Contracts Based on Symbolic Execution. In 2022 IEEE International Conference on Big Data (Big Data), 3,225-3,234.
- Ye, J., Ma, M., Lin, Y., Sui, Y., and Xue, Y. (2020). Clairvoyance: Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings, 274-275.
- Zhou, E., Hua, S., Pi, B., Sun, J., Nomura, Y., Yamashita, K., and Kurihara, H. (2018). Security assurance for smart contract. In 2018 9th IFIP International Conference on New Technologies. Mobility and Security (NTMS), 1-5.