UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE
INGENIERÍA DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

UN ESTUDIO SOBRE LA SEGURIDAD DE ETHEREUM: SUPERANDO LAS BRECHAS EDUCATIVAS MEDIANTE EL DESARROLLO DE FAILLAPOP

GRADO EN INGENIERÍA TELEMÁTICA

MARCO LÓPEZ GONZÁLEZ
MÁLAGA, 2024

E.T.S. de Ingeniería de Telecomunicación, Universidad de Málaga

A Study on Ethereum Security: Bridging Educational Gaps through the Development of Faillapop

Author: Marco López González

Supervisor: Isaac Agudo Ruiz

Department: Languages and Communication Sciences

Degree: Degree in Telematics Engineering

Keywords: Blockchain, Ethereum, Security, Vulnerabilities, Smart Contracts, Educational Resources

Abstract

This work explores Blockchain Technology, focusing on Ethereum, its foundational principles, security concerns and the development of innovative educational resources. The primary objectives are to provide a comprehensive understanding of Ethereum's architecture, investigate its unique security challenges, and introduce Faillapop as an educational tool to enhance security awareness among developers and auditors. The research combines a literature review, practical engagement with Capture The Flag (CTF) challenges, analysis of common vulnerabilities in real-world projects, and the design and development of Faillapop. In addition, this project resulted in the publication and presentation of the paper "*Análisis de las Vulnerabilidades en Smart Contracts: desafíos CTF para mejorar la seguridad*" at the 2023 JITEL in Barcelona. Key findings indicate that traditional educational methods fall short in preparing individuals for real-world security challenges. Faillapop addresses this gap by simulating complex, real-world scenarios, fostering critical thinking and hands-on expertise in identifying and mitigating common Smart Contract (SC) vulnerabilities. Future directions include keeping Faillapop updated with the latest security practices, promoting its widespread use, and conducting cybersecurity workshops to enhance Ethereum's security and contribute to the broader blockchain ecosystem's resilience.

E.T.S. de Ingeniería de Telecomunicación, Universidad de Málaga

Un Estudio sobre la Seguridad de Ethereum: Superando las Brechas Educativas mediante el Desarrollo de Faillapop

Autor: Marco López González

Tutor: Isaac Agudo Ruiz

Departamento: Lenguajes y Ciencias de la Computación

Titulación: Grado en Ingeniería Telemática

Palabras clave: Blockchain, Ethereum, Seguridad, Vulnerabilidades, Contratos Inteligentes, Recursos Educativos

Resumen

Este trabajo explora la tecnología blockchain, centrándose en Ethereum, sus principios fundacionales, problemas de seguridad y el desarrollo de recursos educativos innovadores. Los objetivos principales son proporcionar una comprensión global de la arquitectura de Ethereum, investigar sus desafíos de seguridad e introducir Faillapop como herramienta educativa para mejorar la conciencia en seguridad entre desarrolladores y auditores. La investigación combina una revisión de la literatura, la participación en desafíos de CTF, el análisis de vulnerabilidades en proyectos reales, y el diseño y desarrollo de Faillapop. Además, este proyecto resultó en la publicación y presentación del artículo "Análisis de las Vulnerabilidades en Smart Contracts: desafíos CTF para mejorar la seguridad" en las Jornadas de Ingeniería Telemática (JITEL) de 2023 en Barcelona. Las principales conclusiones indican que los métodos educativos tradicionales no preparan adecuadamente a los profesionales para los retos de seguridad del mundo real. Faillapop aborda esta carencia mediante la simulación de escenarios complejos, fomentando el pensamiento crítico y la experiencia práctica en la identificación y mitigación de vulnerabilidades en los contratos inteligentes. En el futuro, Faillapop se actualizará con las últimas prácticas de seguridad, se promoverá su uso y se organizarán talleres de ciberseguridad para mejorar la seguridad de Ethereum y contribuir a la resiliencia del ecosistema blockchain.

A mi abuelo,
que con su amor y recuerdo
sigue siendo mi inspiración cada día

Marco López González

Agradecimientos

Me gustaría comenzar agradeciendo a mi familia por todo el cariño y apoyo que me han brindado incondicionalmente. Sin su constante presencia y aliento, superar los obstáculos y alcanzar mis objetivos habría sido mucho más difícil. Gracias por creer siempre en mí y por estar ahí en cada paso del camino.

Un agradecimiento muy especial a mi novia, quien ha sido mi refugio durante todo este proceso. Gracias por tu paciencia, por soportar mis momentos de estrés y por hacer que los días difíciles fueran más llevaderos con tu amor y compañía. A mis amigos, gracias por escucharme, por animarme y por hacer que este camino haya sido mucho más divertido y llevadero. Vuestro apoyo ha sido fundamental.

También quiero extender un profundo agradecimiento a mi tutor, Isaac Agudo. Su gran labor educativa y su constante empuje para que siempre diera lo mejor de mí han sido invaluables. Gracias por tus consejos y por estar siempre dispuesto a ayudar.

Por último, agradecer a José Carlos Ramírez. Su pasión por la ciberseguridad es realmente contagiosa, y su guía y tutoría han sido esenciales para mi crecimiento en este campo. Gracias por inspirarme y por compartir tus conocimientos con tanta generosidad. Sin duda, ambos habéis sido pilares fundamentales en mi evolución académico-profesional, y me ilusiona seguir colaborando y aprendiendo junto a vosotros en el futuro.

Acronyms

SC	Smart Contract
MEV	Maximum Extractable Value
DApps	Decentralized Applications
EVM	Ethereum Virtual Machine
P2P	Peer-to-Peer
EOA	Externally Owned Accounts
CTF	Capture The Flag
MUBs	Machine Unauditable Bugs
ECDSA	Elliptic Curve Digital Signature Algorithm
DEX	Decentralized Exchange
DeFi	Decentralized Finance
JITEL	Jornadas de Ingeniería Telemática

Contents

List of Figures

Part I

Introduction

Chapter 1

Overview and Motivation

Content

Synopsis

The introduction section of this dissertation sets the stage for a comprehensive exploration of Blockchain Technology, with a particular focus on Ethereum and its security challenges. It provides the foundational context, outlines the motivation behind the development of the Faillapop project, specifies its objectives, details the methodological approach used and gives a roadmap of the dissertation.

1.1 Motivation

Blockchain Technology, despite its yet-to-be-established reputation, is often touted as a key driver of society's digital transformation. It has the potential to revolutionize diverse sectors, from managing public and private documents to ensuring traceability across various industries and facilitating digital currencies. By enabling consensus over a permissionless, decentralized network, blockchain emerges as a transformative technology in fields such as FinTech [1], healthcare [2], Internet of Things [3], and supply chains [4].

At the core of blockchain's appeal is its integration of Peer-to-Peer (P2P) systems, cryptographic techniques, pseudonymity, and distributed consensus schemes. These features ensure that confirmed transactions are public, traceable, and tamper-resistant. One of the most prominent implementations is Ethereum, which handles millions of transactions daily [5].

Ethereum operates as a deterministic yet practically unbounded state machine, featuring a globally accessible singleton state and a virtual machine that applies changes to this state. This structure underpins Ethereum's role as a decentralized computing infrastructure, executing programs known as SCs. It uses blockchain to synchronize and store state changes and employs a cryptocurrency called Ether to measure and limit execution resource costs.

A key challenge in blockchain ecosystems is their unique properties, such as immutability, determinism, pseudo-randomness, transparency, and Maximum Extractable Value (MEV), that present both opportunities and security challenges. While these are not vulnerabilities per se, neglecting them during programming can create security loopholes. For instance, in telecommunications, ignoring the inherent broadcast nature of radio waves can lead to communication interceptions. Similarly, disregarding blockchain's transparency can lead to MEV attacks, underscoring the importance of understanding and adapting to the environment.

As discussed in the "Ethereum Security" chapter, Ethereum's decentralized nature necessitates robust security measures to mitigate threats such as MEV attacks, and vulnerabilities in SC logic. These issues have led to substantial financial losses and undermined trust in Blockchain Technology's reliability and security, as evidenced by numerous attacks, such as the one against Euler Finance [6], where vulnerabilities in the protocol's SCs resulted in a $197 million embezzlement. Research [7] highlights that in the first half of 2022, despite $14 million paid as incentives to ethical hackers, over $245 million was lost to real attacks. This disparity underscores the insufficient supply of human auditor resources, which could be mitigated through enhanced educational tools.

Furthermore, the recent boom in the Decentralized Finance (DeFi) industry has attracted a new wave of developers, who often face rapidly evolving platform features. As the ecosystem grows, it is imperative for developers to adapt and evolve their approaches to secure SCs. The motivation behind developing the Faillapop project and undertaking this dissertation is to address these pressing security challenges within Ethereum.

This study [8] aims to gain insights into how SC developers perceive and address security in their projects. Findings suggest that merely accessing documentation, reference implementations, and security tools is insufficient for helping developers identify SC security vulnerabilities. Education that enhances developers' knowledge, motivation, and awareness regarding SC security is crucial. Although educational materials are available, there is a lack of hands-on exercises or labs for SC security. Educational resources should be tailored based on developers' expertise, skills, years of experience, and development stages.

Despite the availability of introductory educational content on Ethereum vulnerabilities and security issues, there is a clear lack of advanced educational resources. Learning from the experience of exploited protocols seems to be the only advanced educational tool. This gap motivated us to develop Faillapop, which mirrors real-world, complex applications, providing students with a realistic and comprehensive learning experience.

In conclusion, the motivations behind this dissertation are multifaceted. They encompass the need to address security gaps, enhance educational resources, and contribute to the broader understanding of blockchain and Ethereum security. By developing Faillapop, we aim to provide a valuable educational tool that prepares developers and auditors to navigate the complex landscape of Ethereum security. Ultimately, this will fortify the platform against vulnerabilities and ensure its secure, transformative potential is realized.

1.2 Objectives

The primary objectives of this work are as follows:

- Provide a through exploration of blockchain fundamentals, Ethereum's architecture, and the underlying principles that ensure its functionality.
- Investigate Ethereum's unique security challenges, focusing on vulnerabilities within SCs and their implications for blockchain security as a whole.
- Assess the efficacy of Faillapop as an innovative educational resource designed to enhance Ethereum security awareness among developers and auditors, preparing them to identify and address vulnerabilities in complex applications.

Traditional educational approaches often fall short in adequately preparing developers and auditors to navigate the complexities of blockchain security. Existing tools and exercises, such as capture-the-flag (CTF) simulations, while useful, often lack the realism and intricacies of real-world blockchain applications. Faillapop aims to bridge this gap by simulating a comprehensive Decentralized Applications (DApps) environment composed of interconnected SCs.

Faillapop provides a realistic training ground where participants engage with complex scenarios, including intentional vulnerabilities, to develop critical thinking and hands-on expertise in identifying, understanding, and mitigating security risks specific to Ethereum SCs. This approach aims not only to improve the skills of blockchain professionals, but also to foster a community sense of improving security standards across the industry.

Moreover, Faillapop seeks to serve as an educational resource for academic institutions looking to enrich their blockchain curricula and for companies aiming to upskill their development and security auditing teams.

In essence, this dissertation and the Faillapop project seek to fortify Ethereum's platform by equipping stakeholders with the knowledge and tools needed to mitigate vulnerabilities, ensuring that Blockchain Technology realizes its full potential as a secure, transparent, and transformative platform in the digital era.

1.3 Methodological Approach

The methodology employed in this work followed a structured approach to achieve a comprehensive understanding and enhancement of Ethereum security through the development and refinement of the Faillapop project. The steps involved are outlined below:

Literature Review and State-of-the-Art Analysis

The initial phase involved an extensive literature review and state-of-the-art analysis of Ethereum, with a particular focus on security aspects. This review encompassed current literature, research papers, and case studies detailing known vulnerabilities, attack vectors, and security best practices within the ecosystem. This foundational research provided a critical baseline for understanding the landscape of Ethereum security and identifying areas where educational tools could be most impactful, establishing the basis for the subsequent development and refinement of the Faillapop project.

Initial Project Audit and CTF Participation

Following the literature review, an audit of the initial state of the Faillapop project was conducted. This audit aimed to assess the existing architecture, identify any preliminary vulnerabilities, and evaluate the project's alignment with current security standards.

To gain practical insights and understand how security challenges are addressed in Ethereum, participation in three of the most popular Web3 CTF competitions was undertaken. These challenges are designed to test participants' skills in identifying and mitigating security vulnerabilities in Ethereum SCs. This hands-on experience was crucial for identifying gaps in existing educational resources and understanding the specific needs that Faillapop could fulfill. Additionally, a paper [9] based on this research was presented and published at the 2023 JITEL in Barcelona.

Research on Common Vulnerabilities in Real Protocols

A thorough investigation into the most common vulnerabilities affecting real-world Ethereum protocols was conducted. This research aimed to ensure that Faillapop would not only mimic the architecture of actual blockchain applications but also incorporate prevalent security flaws. By doing so, the project can provide realistic scenarios for users to practice identifying and mitigating vulnerabilities, thereby enhancing their understanding and skills in Ethereum security.

Adherence to Best Practices

Faillapop was not only designed to incorporate common vulnerabilities but also to follow all best practices recommended by the Ethereum and Solidity communities. This included rigorous testing of SCs, adherence to secure coding standards, and the use of established frameworks like the Foundry Framework for robust development and testing. Ensuring that Faillapop was built on a solid foundation of best practices enhanced its reliability and educational value.

Design and Implementation of New Features

To enhance the educational value of Faillapop and simulate real-world scenarios more accurately, several new functionalities were designed and implemented. This included the development of new SCs such as CoolNFT and PowersellerNFT, which introduce advanced features and use cases relevant to current blockchain applications. Additionally, an ERC1967 Proxy was implemented to enable upgradability in the Shop contract, reflecting the need for maintaining and updating SCs in real-world applications. A commit-reveal scheme for dispute voting was integrated to ensure fair and secure resolution processes within the Faillapop platform. And a comprehensive documentation was created to support these new features, providing detailed guidance and insights into their design, implementation, and security considerations.

In summary, the methodology adopted in this work was a multi-faceted approach combining theoretical research and practical engagement. By integrating insights from academic literature, real-world vulnerabilities, hands-on experience with CTF challenges, and the design of new features, this research aimed to create a comprehensive and effective educational tool for Ethereum security, addressing both the theoretical and practical aspects of Blockchain Technology.

1.4 Dissertation Structure

The present document is structured into four main parts, each designed to systematically explore and address the research objectives. The structure ensures a logical progression from theoretical foundations to practical applications and evaluations.

Part I: Introduction

The first part lays the groundwork for the dissertation. It begins with a Motivation, detailing the importance and relevance of the work. The Objectives section outlines the specific goals of the research, followed by a detailed Methodological Approach explaining the steps taken to achieve these goals. It concludes with this Dissertation Structure, giving readers a roadmap of the work.

Part II: State of the Art

The second part provides a thorough examination of the current state of blockchain and Ethereum technologies. It starts with an Introduction to blockchain, explaining its foundational principles and mechanisms. This is followed by a focused discussion on Ethereum, highlighting its unique features and capabilities. The section on Ethereum Security compares traditional software security with the unique challenges of Web3, examines common SC vulnerabilities, and explores methods and tools for detecting them.

Part III: Project Development

The third part is the heart of the dissertation, detailing the development of the Faillapop project. It begins with the Origin and Objectives of Faillapop, explaining the rationale behind its creation and its intended goals. The Initial Security Evaluation assesses the security landscape of the project at its inception. The chapter on Protocol Development and Enhancements describes the evolution of Faillapop, including the deliberate introduction of vulnerabilities to simulate real-world scenarios and enhance its educational value. This part highlights the practical aspects of the research, demonstrating how theoretical insights were applied.

Part IV: Conclusion

The final part evaluates the security challenges inherent in Ethereum and Blockchain Technology. It underscores the necessity for comprehensive educational resources like Faillapop, designed to simulate real-world scenarios and equip developers with practical skills to mitigate risks effectively. This section also discusses potential future work, ensuring Faillapop remains a pivotal tool for fostering secure practices and resilience in Ethereum applications through ongoing education and community engagement initiatives.

Part II

State of the Art

Chapter 2

Blockchain and Ethereum

Content

Synopsis

In this chapter we delve into the foundational concepts and technologies that underpin blockchain systems, with a special focus on Ethereum. It begins with an introduction to Blockchain Technology, outlining its essential features and mechanisms. The chapter then explores various consensus mechanisms, comparing the methodologies of Bitcoin and Ethereum to highlight their unique approaches to achieving distributed consensus.

A detailed examination of Ethereum follows, starting with its Cryptographic foundations, Ethereum accounts, Transactions and an in-depth analysis of the Ethereum Virtual Machine (EVM), which executes SCs. Then, SCs, their development in Solidity, and the role of oracles in integrating external data into the blockchain are thoroughly discussed.

The insights from this chapter provide a comprehensive understanding of the technological and operational aspects of Ethereum, setting the stage for a deeper investigation into its security challenges and the educational tools developed to address them in subsequent chapters.

2.1 Intro

Blockchain Technology [10] gained special attention after the publication of Satoshi Nakamoto's white paper in 2008 [11]. In the white paper, Satoshi provided a solution to the double-spending problem for digital currency within a decentralized P2P network [12]. Although blockchain and its numerous applications have recently come to the forefront, many of its underlying technologies have been under research since the 1980s. Ralph Merkle introduced the concept of Merkle trees in 1979, demonstrating their use in public key distribution and digital signatures [13]. Subsequently, in 1982, David Chaum proposed a vault system for establishing, maintaining and trusting mutually suspicious groups [14]. In 1991, Stuart Haber and W. Scott Stornetta discussed a cryptographically secure chain of blocks and timestamping of digital components in their article, later integrating their concept with Merkle Trees [15]. Nick Szabo introduced SCs in 1997 [16]. Around the same time, other developments propelled the Blockchain concept: Napster [17] introduced in 1999, facilitated P2P music sharing. Later, Adam Back incorporated Proof of Work in the hashcash algorithm [18], and in 2004, Hal Finney introduced reusable PoW [19].

Despite its yet-to-be-established reputation as a technology, blockchain is often touted as a key driver of the digital transformation of society, impacting areas ranging from the management of public and private documents to traceability in various industries and digital currencies. By enabling consensus over a permissionless decentralized network [20], blockchain emerges as a disruptive technology across multiple fields, including FinTech [1], healthcare [2], the Internet of Things [3], and supply chains [4].

2.2 Blockchain

Blockchain is a self-governed P2P network transaction system that enables the secure execution of operations without the need for a trusted third party [21]. Transactions are conducted on a decentralized ledger composed of sequential blocks, each linked to its predecessor with an immutable connection, ensuring the chain integrity. Each block stores validated transactions according to consensus algorithms. The ledger is shared and replicated among network participants, allowing them to read and write data transparently.

The components of an open, public blockchain are:

- A **P2P network** connecting participants and propagating transactions and blocks of verified transactions, based on a standardized “gossip” protocol.
- **Messages**, in the form of transactions, representing state transitions.
- A set of **consensus rules**, governing what constitutes a transaction and what makes for a valid state transition.
- A **state machine** that processes transactions according to the consensus rules.
- A **chain of cryptographically secured blocks** that acts as a journal of all the verified and accepted state transitions.
- A **consensus algorithm** that decentralizes control over the blockchain, by forcing participants to cooperate in the enforcement of the consensus rules.
- A game-theoretically sound **incentivization scheme** to economically secure the state machine in an open environment.
- One or more **open source software implementations** of the above, what is known as clients.

The design of these components can vary, leading to the creation of different types of blockchains.

What makes BT appealing is the combination of P2P systems, cryptographic techniques, distributed consensus schemes and pseudonymity. These elements ensure that the set of confirmed transactions becomes public, traceable and tamper-resistant. The latter property is achieved by linking subsequent blocks using cryptographic hash functions, as shown in Figure 2.1. Any modification of transaction data in a block B_i would alter the hash contained in the subsequent block B_{i+1}, thus changing the content of the block B_{i+1} and so on. The blockchain is replicated across multiple nodes in a P2P fashion, making any attempt to alter it easily detectable due to inconsistencies among all replicas.

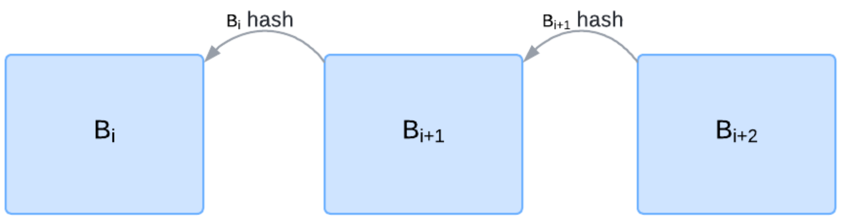


Figure 2.1: Each block is linked to its predecessor by its hash

2.2.1 Consensus Mechanisms

In Blockchain Technology, consensus refers to the automatic matching of states in multiple processes, not an agreement between individuals. This computer science concept ensures that the states of replicated state machines across multiple computers always match. This synchronization allows the system to function even if some computers fail, as correct records are maintained by comparing replicas.

Consensus addresses the broader challenge of synchronizing state in distributed systems, ensuring that different participants agree on a single system-wide state without a central decision-making entity. However, this decentralization requires reconciliation through other means, primarily consensus algorithms, which balance security and decentralization.

Proof of Work

In a Proof-of-Work (PoW) system, consensus nodes validate transactions and create new blocks by solving complex mathematical puzzles. This process requires significant computational power, making it energy-intensive. Miners compete to solve these puzzles, and the first to do so gets to add the new block to the blockchain and receive a reward. This system is secure because altering the blockchain would require redoing the PoW for all subsequent blocks, which is computationally impractical.

Proof of Stake

In a Proof-of-Stake (PoS) system, consensus nodes are chosen based on the amount of cryptocurrency they hold and are willing to "stake" as collateral. The ability to validate transactions and create new blocks is proportional to the amount of stake held. The more cryptocurrency a staker holds, the more likely they are to be selected for validation and block creation. Validators are incentivized to act in the network's best interest because any attempt to compromise the system would result in a loss of their staked cryptocurrency.

2.2.2 Bitcoin & Ethereum Overview

Initially, people recognized the power of the Bitcoin model and sought to move beyond cryptocurrency applications. However, developers faced a conundrum: they either had to build on top of Bitcoin or start a new blockchain. Building upon Bitcoin meant adhering to the network's intentional constraints and finding workarounds. The limited set of transaction types, data types, and storage sizes seemed to restrict the kinds of applications that could run directly on Bitcoin. Anything beyond these constraints required additional off-chain layers, which negated many advantages of using a public blockchain. For projects needing more freedom and flexibility while staying on-chain, creating a new blockchain was the only viable option, leading to the development of Ethereum.

You can think of Bitcoin as a distributed consensus state machine where transactions cause a global state transition, altering the ownership of coins. Ethereum, on the other hand, is also a distributed state machine but tracks the state transitions of a general-purpose data store. This data store can hold any data expressible as a key-value tuple, similar to the data storage model of Random Access Memory (RAM) used by most general-purpose computers. Ethereum has memory that stores both code and data, using the Ethereum blockchain to track changes over time. Like a general-purpose stored-program computer, Ethereum can load code into its state machine, run that code, and store the resulting state changes in its blockchain. Two critical differences from most general-purpose computers are that Ethereum state changes are governed by consensus rules and the state is distributed globally.

2.3 Ethereum

Blockchain Technology has surged in popularity over the years due to its immutability and transparency within a permissionless and decentralized environment. One of the most well-known and widely used implementations is Ethereum. As of May 30, 2024, Ethereum's cryptocurrency market capitalization surpassed $450 billion, with millions of transactions being executed daily [22] [5].

Ethereum is a deterministic yet practically unbounded state machine, consisting of a globally accessible singleton state and a virtual machine that applies changes to that state. Thus, it acts as a globally decentralized computing infrastructure that executes programs called SCs. It uses a blockchain to synchronize and store the system's state changes, along with a cryptocurrency called ether to meter and constrain execution resource costs.

The platform enables developers to build DApps with built-in economic functions, providing high availability, auditability, transparency, and neutrality.

Ethereum's groundbreaking innovation is its ability to combine the general-purpose computing architecture of a stored-program computer with a decentralized blockchain, thereby creating a distributed single-state world computer.

It offers the ability to execute programs known as SCs, which are written in the high-level programming language Solidity. These contracts are immutably stored on the blockchain and facilitate the execution of predefined terms without consulting third parties, in a pseudo-anonymous, transparent, and tamper-resistant manner. Although SCs can be made updatable through the use of a proxy that routes calls to a new implementation, the original contract remains published on the blockchain, maintaining its immutability [23].

Ethereum nodes send and receive messages of transactions and blocks over the P2P network to achieve distributed consensus. The operational stability of a blockchain system is influenced by the message forwarding protocol, peer discovery protocol, and the topology of its underlying P2P network. Therefore, it is crucial to analyze and understand the P2P networks of blockchain systems.

This work [24] presents the Ethereum Network Analyzer (Ethna), a tool that probes and analyzes the P2P network of the Ethereum blockchain. Ethna measured and analyzed the degree distribution of Ethereum nodes according to the random selection feature of the message forwarding protocol in the Ethereum P2P network. The findings revealed that the average degree of Ethereum P2P network nodes is 47, with a few super nodes having very high degrees. The average delay for broadcasting a transaction across the entire Ethereum P2P network is around 200 ms. It takes 3-4 hops to broadcast a new block or transaction to the whole network, indicating that the Ethereum P2P network conforms to the small-world property.

2.3.1 Cryptography

Contrary to common belief, the Ethereum protocol does not involve encryption. Encryption transforms data so only those with the key can decrypt it. In Ethereum, all communications with the platform and between nodes are unencrypted and readable by anyone. This transparency ensures everyone can verify the correctness of state updates and reach consensus.

Many people mistakenly believe blockchain inherently protects privacy, but this is not its primary purpose. Advanced cryptographic tools, such as zero-knowledge proofs [25] and homomorphic encryption [26], are being developed to enable encrypted calculations on the blockchain while still allowing consensus in the future.

Elliptic Curve Cryptography

Elliptic curve cryptography (ECC) is fundamental to Ethereum's use of private keys and digital signatures [27]. Ethereum employs public key cryptography to create a public-private key pair, representing an account with a publicly accessible handle and private control.

When a transaction is sent to the Ethereum network to move funds or interact with SCs, it must include a digital signature created with the corresponding private key. The elliptic curve mathematics allows anyone to verify the transaction's validity by checking that the digital signature matches the transaction details and the Ethereum address being accessed.

Key Generation and Management

Generating keys begins with finding a secure source of entropy. Creating an Ethereum private key involves selecting a number in the interval $[1, 2^{256}]$. The public key is then calculated from the private key using elliptic curve multiplication, resulting in another point on the curve, which serves as the public key. Finally, the Ethereum address is derived from the last 20 bytes of the Keccak-256 hash [28] of the public key.

Private keys are never transmitted or stored on Ethereum. They should remain private and never appear in network messages or on-chain data. Only account addresses and digital signatures are transmitted and stored on the Ethereum system.

To include a transaction in the blockchain, it must have a valid digital signature. As long as users keep their private keys secure, these digital signatures prove ownership of the funds. Users manage their keys and sign transactions using wallets, which serve as the primary user interface to Ethereum. These software applications help users manage their keys and facilitate secure transactions.

Elliptic Curve Digital Signature Algorithm (ECDSA)

Ethereum utilizes the ECDSA for digital signatures, which serve three primary purposes:

1. **Authorization**: Proves the owner of the private key has approved the transaction.
2. **Non-repudiation**: Ensures that the authorization cannot be denied.
3. **Integrity**: Confirms that the transaction data has not been altered post-signature.

Digital signatures in Ethereum consist of two parts: an algorithm to create the signature using a private key and message, and an algorithm to verify the signature using the message and public key. In Ethereum, the message being signed is the Keccak-256 hash of the RLP-encoded transaction data.

2.3.2 Ethereum Accounts

Ethereum accounts are the basic elements of the Ethereum network, each comprising four fields: *nonce*, *balance*, *storage*, and *code*. The *nonce* is a transaction counter, incremented with every new transaction sent by the account. The *balance* indicates the amount of Ether the account holds. *Storage* serves as memory space for code and its execution, while the *code* field stores the SC code [29].

There are two types of accounts:

- **Externally Owned Accounts (EOA)**
- **Contract Accounts**

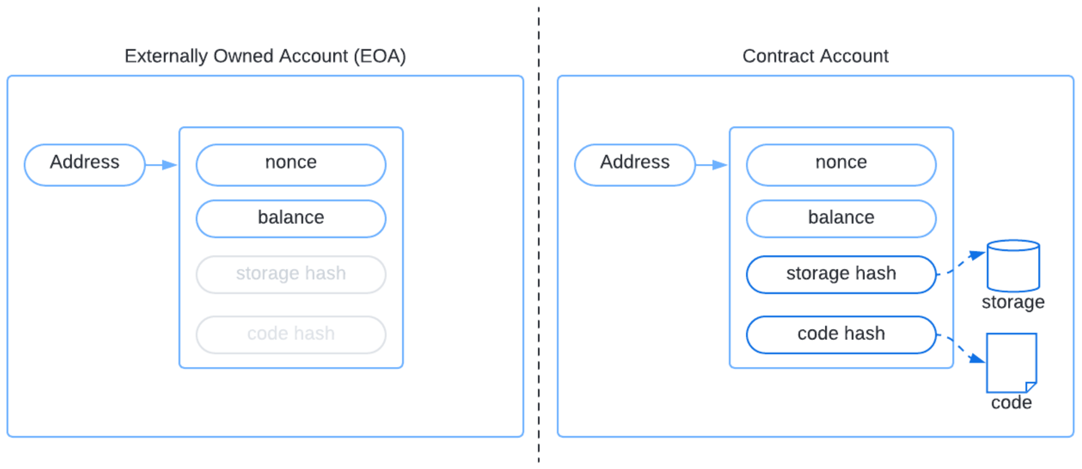


Figure 2.2: Contract accounts contain EVM code and storage

As illustrated in Figure 2.2, both are identified by an Ethereum address, and the key difference lies in the code and storage fields. EOAs are controlled by public-private key pairs and do not contain code, whereas contract accounts are governed by their code [29] [30]. Both types of accounts are hashed and stored in a data structure known as a modified Merkle Patricia tree [30], which has its root hashed and stored in every block.

EOAs can initiate actions that alter the state of the EVM, known as transactions. In contrast, contract accounts cannot initiate transactions due to their lack of private keys. However, when a transaction is directed to a contract address, it can react to it by calling other contracts, building complex execution paths.

2.3.3 Transactions

Transactions in Ethereum are signed messages created by EOA. As a global singleton state machine, Ethereum relies on transactions to alter its state, as we can see in Figure 2.3.

Transaction propagation begins with the originating Ethereum node creating or receiving a signed transaction, which is then validated and transmitted to other directly connected nodes. Each node typically maintains connections to at least 13 other nodes, known as its neighbours. As each neighbour node receives the transaction, they validate it, store a copy, and propagate it to their neighbours. This process ensures that within seconds, an Ethereum transaction reaches nodes across the globe.

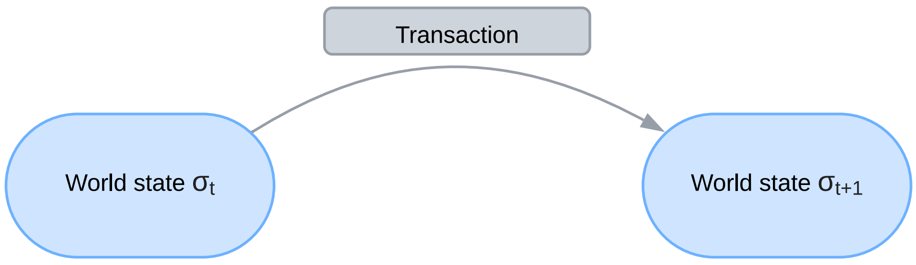


Figure 2.3: Transactions represent valid transitions between states

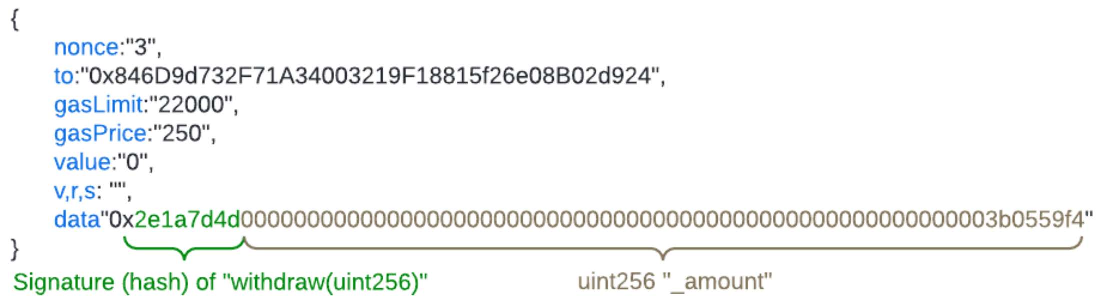
Transactions are atomic: they either execute entirely or not at all. All changes to the global state are recorded only if the transaction completes successfully. If execution fails, all changes are reverted as if the transaction never occurred. However, even failed transactions are recorded as attempts, with the ether spent on gas deducted from the originating account.

A transaction is a serialized binary message that contains the following data:

- **Nonce**: A sequence number, issued by the originating EOA, used to prevent message replay.
- **Gas price**: The price of gas the originator is willing to pay
- **Gas limit**: The maximum amount of gas the originator is willing to buy
- **Recipient**: The destination Ethereum address
- **Value**: The amount of ether to send to the destination
- **Data**: The variable-length binary data payload
- **v, r, s**: The three components of an ECDSA digital signature of the originating EOA.

The data payload includes a hex-serialized encoding of a function selector—the first 4 bytes of the Keccak-256 hash of the function's prototype— along with the function arguments. This payload is interpreted by the EVM as a contract invocation.

Figure 2.4 illustrates the body of a transaction that is intended to call the function `withdraw(uint256 _amount)` in a contract deployed at the account address `0x846D9d732F71A34003219F18815f26e08B02d924`. The transaction is being sent from an address with a nonce of 3. The fields `v`, `r`, and `s`, which represent the components of the ECDSA digital signature, are empty because this is the transaction body before being signed. The transaction includes a gas limit of 22,000 units, a gas price of 250 wei[1], and a value of 0 ether. The `data` field contains the encoded function call with the parameter `_amount` set to `0x3b0559f4`.



```
{
    nonce:"3",
    to:"0x846D9d732F71A34003219F18815f26e08B02d924",
    gasLimit:"22000",
    gasPrice:"250",
    value:"0",
    v,r,s: "",
    data"0x2e1a7d4d000000000000000000000000000000000000000000000000000000003b0559f4"
}
```

Figure 2.4: Example of a transaction to call a contract function

A special type of transaction is one that creates a new contract on the blockchain. These transactions are sent to a special destination address known as the zero address and must contain a data payload with the compiled bytecode for the contract. Such transactions only create the contract, though they can also include an ether amount in the value field to provide the new contract with an initial balance. If the contract being created has a constructor function, it will be executed to initialize the contract's state.

2.3.4 Ethereum Virtual Machine

The EVM is responsible for deploying and executing SCs on the Ethereum network. As shown in Figure 2.5, it features a stack-based architecture with a 256-bit word size and several addressable data components:

- **Immutable program code ROM**: Contains the bytecode of the SC to be executed.
- **Volatile memory**: Initialized to zero for temporary data storage during execution.
- **Permanent storage**: Part of the Ethereum state, also zero-initialized.

The EVM lacks intrinsic scheduling capabilities; instead, Ethereum clients manage execution order by processing verified block transactions to determine

[1]Wei is the smallest denomination of ether, the cryptocurrency used on the Ethereum network. 1 ether = 10^{18} wei.

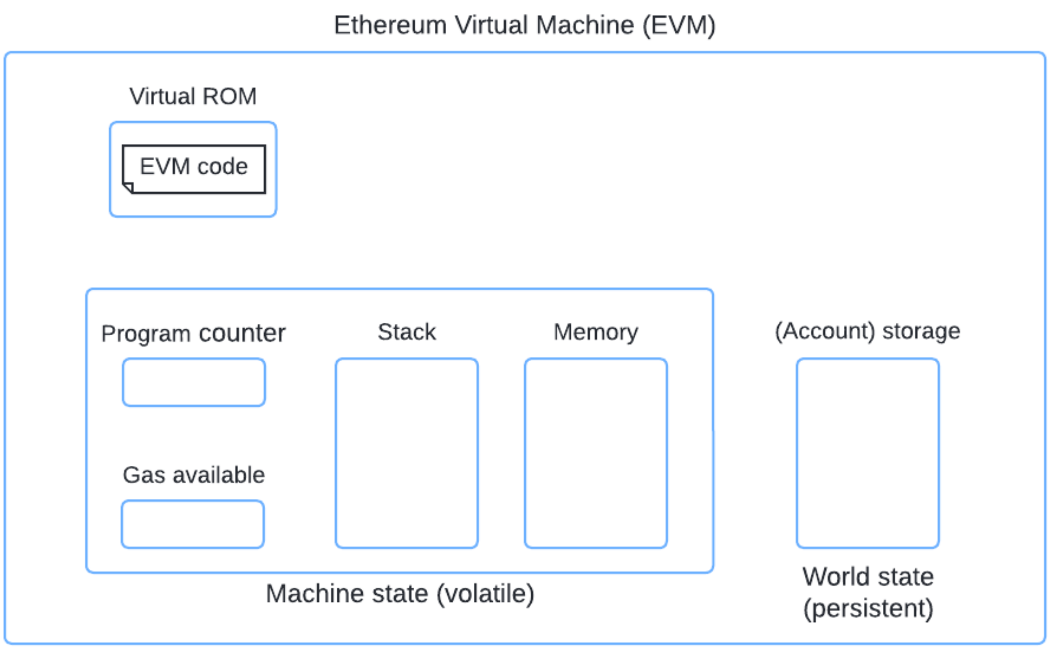


Figure 2.5: The EVM is a simple stack-based architecture

the sequence and necessity of SC execution. Consequently, the Ethereum "world computer" operates in a single-threaded manner.

EVM's primary function, as defined by the Ethereum protocol, is to update the Ethereum state by computing valid state transitions resulting from SC execution. This makes Ethereum a transaction-based state machine.

At the highest level, the Ethereum world state maps Ethereum addresses (160-bit values) to accounts. Each address corresponds to an account, which may hold ether, contract code, and persistent storage. Therefore, transactions trigger updates to Ethereum's state, which are essentially modifications to any of an account's fields.

Gas and Halting Problem

Ethereum's Turing-complete nature introduces complexity, especially in open systems, due to the halting problem, which Alan Turing demonstrated as the impossibility of predicting whether a program will terminate without executing it.

In practical terms, a program might run indefinitely if it contains an infinite loop, whether intentional or accidental. This unpredictability poses a significant challenge for Ethereum, as each node must validate every transaction and execute any associated SC. The EVM cannot determine in advance if a SC will halt or how long it will take to run without actual execution. This situation could be exploited to launch a Denial of Service (DoS) attack by creating a contract that runs indefinitely.

To mitigate this risk, Ethereum employs a gas mechanism. During SC execution, the EVM tracks each instruction, assigning a predetermined gas cost. When a transaction initiates SC execution, it must include an amount of gas that caps the computational effort. If the execution exceeds the provided gas, the EVM halts the process.

Thus, the EVM is a quasi-Turing-complete state machine—"quasi" because execution is limited by the finite amount of gas, ensuring a bounded number of computational steps. This mechanism effectively addresses the halting problem by preventing endless execution.

When a transaction triggers SC execution, an EVM instance is created with the transaction environment, which includes static block information and immutable transaction parameters such as gas price, gas limit and origin account's address. The EVM then loads the contract bytecode into its program code ROM, initializes the program counter to zero, loads the contract's storage, and zeroes out the memory. The gas supply is set to the transaction's gas limit. As execution proceeds, the gas supply depletes according to the operations' gas costs. If gas runs out, an *Out of Gas* (OOG) exception occurs, halting execution and abandoning the transaction. Only the nonce increment and ether deduction for gas are recorded, with no other state changes applied.

The EVM operates in a sandboxed environment, in isolation from Ethereum's main state. If execution fails, the sandbox state is discarded. If successful, the main state is updated to reflect the sandboxed state's changes, including contract storage modifications, new contract creations, and ether transfers.

Contracts can invoke other contracts, instantiating a new EVM instance for each call. Each instance inherits a sandboxed state from its parent instance and receives a specified gas supply. If insufficient gas is provided, the instance may halt with an exception, discarding its state and reverting control to the parent EVM instance.

2.3.5 Smart Contracts

The term SC has evolved over the years to encompass various concepts. In the 1990s, cryptographer Nick Szabo defined SCs as "a set of promises, specified in digital form, including protocols within which the parties perform on the other promises." Today, in the context of Ethereum, SCs refer to immutable computer programs that execute deterministically within the EVM. Once deployed, these contracts cannot be altered, and their outcomes remain consistent across all users, provided the same transaction context and blockchain state.

Typically written in high-level languages like Solidity, SCs are compiled into EVM bytecode [31] [32] and deployed to the Ethereum blockchain through a contract creation transaction, as depicted in Figure 2.6. Upon successful deployment, the contract is assigned a unique contract address.

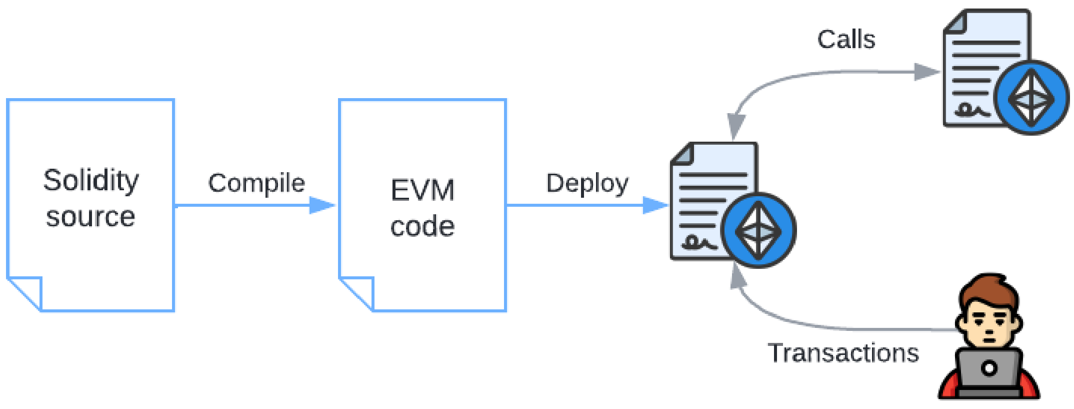


Figure 2.6: Smart Contracts deployment

SCs have diverse applications, from simple agreements between individuals to complex arrangements among businesses and organizations. They can automate various processes, including payment settlements, supply chain management, and digital identity verification.

2.3.6 Solidity

Solidity, created by Dr. Gavin Wood, is a procedural programming language specifically designed for writing SCs on the Ethereum blockchain. Its features are tailored to support execution within the decentralized environment of Ethereum.

The primary product of the Solidity project is the Solidity compiler (solc), which translates Solidity programs into EVM bytecode, as illustrated in Figure 2.7. This

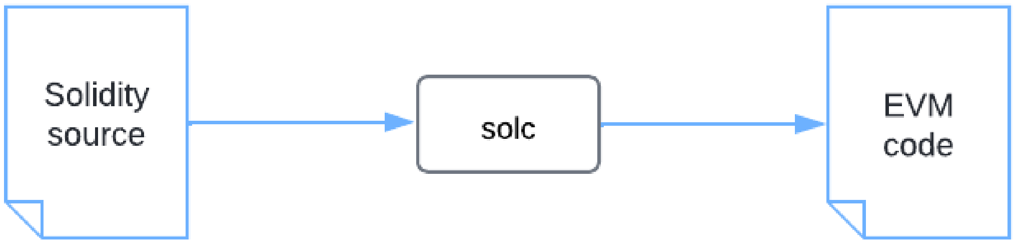


Figure 2.7: EVM code generation

project also oversees the application binary interface (ABI) standard for Ethereum SCs. An ABI defines how data structures and functions are accessed in machine code and is specified as a JSON array of function descriptions and events. Using the ABI, applications can construct transactions that call the functions with the correct arguments and types.

EVM bytecode is akin to assembly language, comprising instructions for stack operations, arithmetic, jumps, and memory access. The EVM's stack-based architecture is enhanced with specific opcodes for blockchain functions, such as the SHA3 hash, environment access, storage manipulation, and internal contract calls. Notably, the SELFDESTRUCT opcode allows for the deletion of a contract, but only after the successful execution of the triggering transaction.

Key Features and Syntax

Solidity's syntax includes several key features designed for secure and efficient SC development:

- **Value Transfers**: Transferring funds between contracts using low-level functions translates to internal call transactions, which implies that calling a contract might also execute code and in particular it can fail because the available gas is not sufficient for executing the code. In addition — as in the EVM — these kind of calls do not enable exception propagation, so that the caller manually needs to checks for the return result. Another special feature of Solidity is that it allows for defining so called fallback functions for contracts that get executed when a call via low-level functions was performed or an address is called that however does not properly specifies the concrete function of the contract to be called.

- **Data Types**: Solidity supports various data types, including:
  - **Boolean**
  - **Integer**: Signed and unsigned integers in 8-bit increments from `int8` to `uint256`
  - **Address**: A 20-byte Ethereum address with functions like balance and call
  - **Fixed and Dynamic Byte Arrays**
  - **Enums, Arrays, Structs, and Mappings**: Mappings act as hash lookup tables for key-value pairs
- **Value Literals**: Solidity includes units for time (`seconds`, `minutes`, `hours`, `days`) and `ether` (`wei`, `finney`, `szabo`, `ether`).

Execution Context

During contract execution in the EVM, Solidity contracts have access to several global objects:

- **Transaction Context**: Provides transaction-related information, such as `tx.gasprice` or `tx.origin`
- **Message Call Context**: Includes details like `msg.sender`, `msg.value`, `msg.gasleft`, `msg.data` and `msg.sig`
- **Block Context**: Contains current block information, including `block.blockhash(blockNumber)`, `block.coinbase`, `block.gaslimit`, `block.number` and `block.timestamp`
- **Address Object**: Offers functionalities like `address.balance`, `address.code`, `address.transfer(amount)`, and low-level functions such as `address.call(payload)` and `address.delegatecall()`

Contract structure

The principal data type in Solidity is the contract, a container for data and methods. Functions within contracts can be called by EOA or other contracts. Functions are declared as follows:

```
function FunctionName([parameters])
↪  {public|private|internal|external} [pure|view|payable]
↪  [modifiers] [returns(return types)]
```

Function visibility determines how and when functions can be accessed:

- **Public**: Callable by other contracts, EOA transactions, or within the contract
- **External**: Callable by other contracts and EOAs, but not from within the contract unless prefixed with `this`
- **Internal**: Accessible only from within the contract or derived contracts
- **Private**: Accessible only within the contract, not by derived contracts

Function behaviour modifiers include:

- **View**: Functions that do not modify the state
- **Pure**: Functions that do not read or write to storage, operating only on input arguments prefixed with `this`
- **Payable**: Functions that accept incoming payments; non-payable functions reject payments

Advanced Features

- **Function Modifiers**: Used to create reusable conditions for functions, such as access control
- **Inheritance**: Allows for modular, extensible, and reusable contract development by extending simple, generic contracts into more specialized ones
- **Error Handling**: Managed by `assert`, `require` and `revert` functions. `assert` and `require` evaluate conditions and stop execution with an error if the condition is false, while `revert` halts execution and reverts any state changes

Events and Logging

When a transaction completes, it generates a transaction receipt containing log entries about actions during execution. Solidity uses events to construct these logs, which are particularly useful for light clients and DApps services to watch for specific events and respond accordingly.

2.3.7 Oracles

For the EVM to maintain consensus, its execution must be entirely deterministic, relying solely on the shared context of the Ethereum state and signed transactions. This determinism has two significant implications: first, the EVM and SCs lack an intrinsic source of randomness; second, external data can only be introduced via the data payload of a transaction.

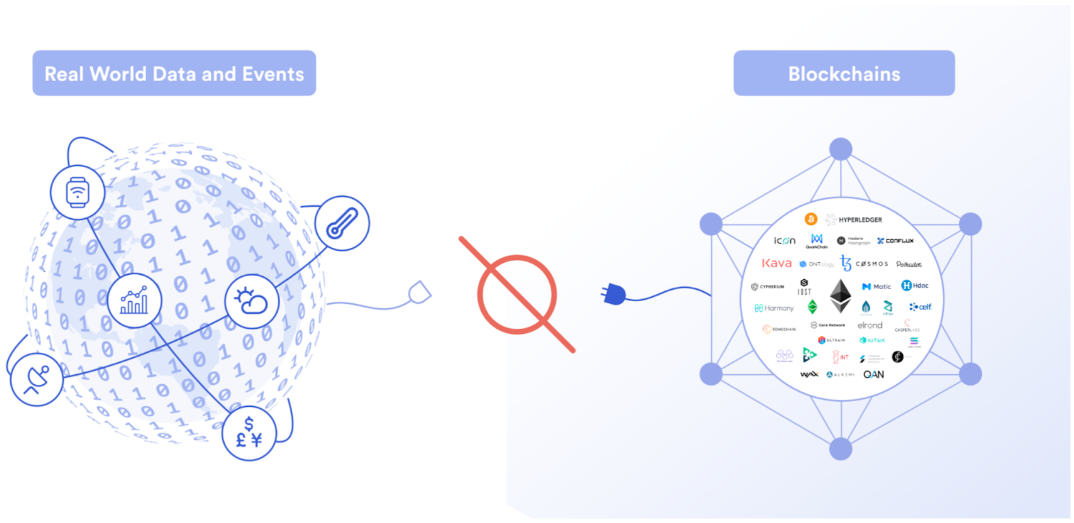
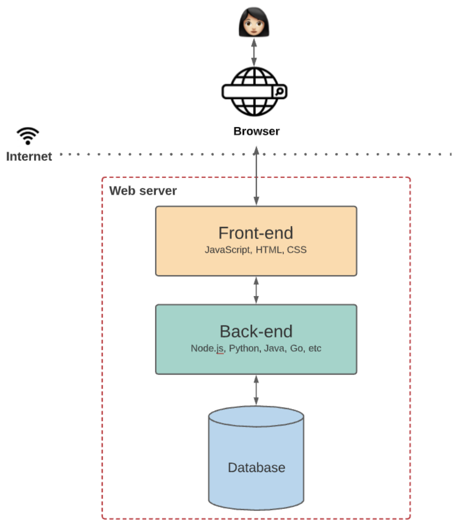


Figure 2.8: Blockchains cannot natively connect to external data and events [33]

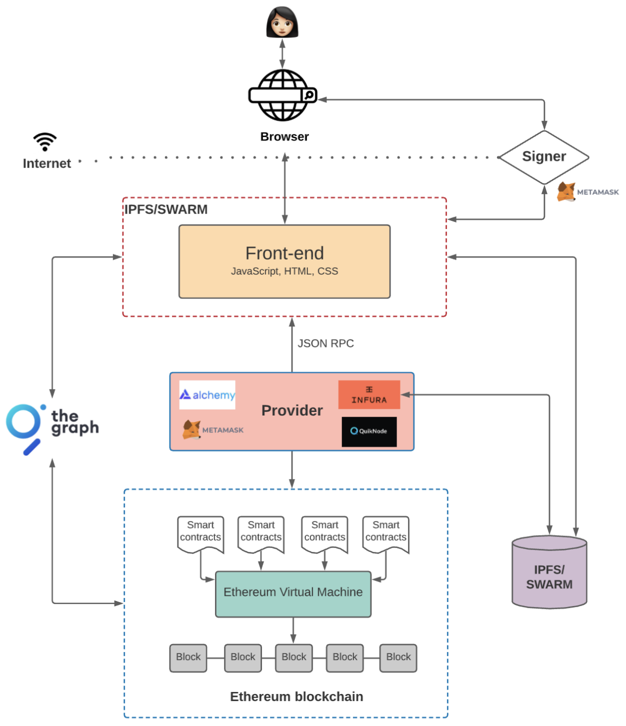
As presented in Figure 2.8, Blockchains have difficulties in obtain external data. This is solved by the use of Oracles that serve as a bridge between the off-chain world and Ethereum, providing a means to securely import external information, such as exchange rates or random numbers, for SCs to use. Ideally, oracles operate in a trustless manner, ensuring the integrity and reliability of the data they provide without compromising the decentralized nature of the blockchain. They are crucial for enabling SCs to interact with real-world data, thereby expanding the range of potential DApps.

2.3.8 Decentralized Applications

DApps are designed to operate in a mostly or entirely decentralized manner, encompassing the backend and frontend software, data storage, message communications, and name resolution. By leveraging Blockchain Technology, DApps offer greater transparency, security, and resilience compared to traditional applications. Figure 2.9 illustrates the differences between the architecture of Web2 applications and DApps.



(a) Web2 App



(b) Web3 App (DApp)

Figure 2.9: Architectural differences between Web2 and Web3 applications

The push for decentralization is driven by the desire to create applications that are resistant to censorship, tampering, and single points of failure. In contexts where trust, transparency, and security are paramount—such as financial services, identity verification, and supply chain management—decentralization can be highly beneficial. It can empower users by giving them control over their data and transactions without relying on intermediaries. In the current Ethereum ecosystem, truly decentralized apps are still relatively rare. Most DApps rely on some degree of centralized services or servers for certain aspects of their operations. However, as the technology matures and infrastructure improves, the trend is moving towards greater decentralization.

Despite that, not all applications benefit from decentralization. For many use cases, the complexity, slower performance, and higher costs associated with decentralized architectures may outweigh the advantages. Centralized systems can often deliver faster, more efficient services that are easier to develop and maintain. For instance, applications that do not require high levels of trust or transparency, such as simple content delivery or basic e-commerce, might not gain significant benefits from being decentralized.

Chapter 3

Ethereum Security

Content

Synopsis

As we transition to the next part of this work, we delve into the critical aspect of security within the Ethereum ecosystem. While the innovations discussed thus far promise a more transparent and resilient digital landscape, they also introduce new vulnerabilities and challenges. The security of Ethereum, its SCs, and the broader decentralized environment is paramount to its success and adoption. In the following sections, we will explore the various security issues inherent to the platform, examining both the threats it faces and the measures that can be taken to mitigate these risks, ensuring a secure and reliable platform for future applications.

3.1 Traditional Software vs Web3

SC development is relatively new compared to traditional software development. While SCs share similarities with traditional software as pieces of code developed by programmers, the nature of attacks can be qualitatively different. In traditional software, security vulnerabilities and functional bugs are largely distinct. Security vulnerabilities usually manifest in limited forms, such as buffer overflows [34], information leaks [35], and privilege escalation[36]. In contrast, SC vulnerabilities are often functional bugs. Due to the unique nature of SCs, incorrect outputs usually indicate monetary loss. Thus, finding these vulnerabilities requires checking domain-specific properties, which is more challenging than identifying a limited set of general security properties in traditional software. However, general software engineering best practices—such as code refactoring, modular structures, good documentation, and state machine diagrams to visualize execution flows—should be applied in SC development to identify and prevent security issues more effectively.

With the recent boom in the DeFi industry, a new wave of developers has joined this space, writing SCs. Blockchain Technologies are evolving dramatically, and developers of decentralized apps often face rapidly changing platform features [37]. As the ecosystem continues to grow, it is imperative that those developers adapt and evolve their approaches to secure SCs, integrating both traditional software best practices and innovative security measures specific to Blockchain Technology. This dual approach will help mitigate risks and enhance the reliability and trustworthiness of DApps.

3.1.1 Intrinsic Properties of Blockchain Ecosystems

The unique characteristics of blockchain ecosystems present both opportunities and security challenges. These properties include immutability, determinism, pseudo-randomness, transparency and MEV. They are not vulnerabilities per se, but failing to account for them when programming can create security loopholes. For example, in telecommunications, ignoring the inherent broadcast nature of radio waves could allow attackers to intercept communications. Similarly, ignoring blockchain's transparency can lead to MEV attacks, highlighting the importance of understanding and adapting to the environment.

Immutability

SCs cannot be changed after deployment [38]. Although updates can be facilitated through a proxy that routes calls to a new implementation, the original contract remains on the blockchain, maintaining its immutability [39]. This feature requires meticulous development practices and awareness of common pitfalls and security best practices.

Determinism

Blockchain transactions and SC executions must be deterministic. This means that given the same inputs and state, the outcome is always the same. This property ensures consistency across the distributed network but also requires developers to handle all potential state transitions and inputs explicitly.

Transparency

All transactions and SC code are publicly visible on the blockchain. This transparency is a double-edged sword: it allows for greater trust and verification but also exposes the inner workings of contracts to potential attackers.

MEV

MEV, or Maximum Extractable Value, refers to the additional value that can be extracted from block production beyond the standard block reward and gas fees by manipulating transaction inclusion, exclusion, and ordering within a block. This process involves validators or bots exploiting the transaction ordering in blockchain networks, such as Bitcoin and Ethereum, where pending transactions reside in the mempool until included in a block.

Blockchain networks are immutable ledgers maintained by decentralized block producers—miners in Proof-of-Work (PoW) blockchains and validators in Proof-of-Stake (PoS) networks. These block producers aggregate pending transactions into blocks, validate them, and append them to the global ledger. While these networks ensure the validity of transactions and continuous block production, they do not guarantee the order in which transactions are included.

Block producers have full autonomy in selecting which transactions from the mempool to include in their blocks. Although they typically prioritize transactions with the highest fees to maximize profits, this is not a network requirement. This autonomy allows them to extract additional value by reordering transactions, creating MEV.

MEV can be exploited through techniques such as front-running, sandwich attacks, and back-running. In a sandwich attack, a malicious actor places a trade just before a victim's pending transaction (front-running) and another immediately after (back-running), artificially inflating the price to generate a profit. The transparency of the mempool makes these attacks feasible, as validators or bots can monitor and manipulate pending transactions.

To mitigate MEV, developers can use cryptographic techniques such as commit-reveal schemes, which conceal transactions from miners until they are included in a block. Implementing randomness in transaction ordering can also reduce predictability, making it harder to exploit MEV. Additionally, projects should be designed to avoid transaction ordering dependencies, and when unavoidable, cryptographic schemes should be used to prevent MEV.

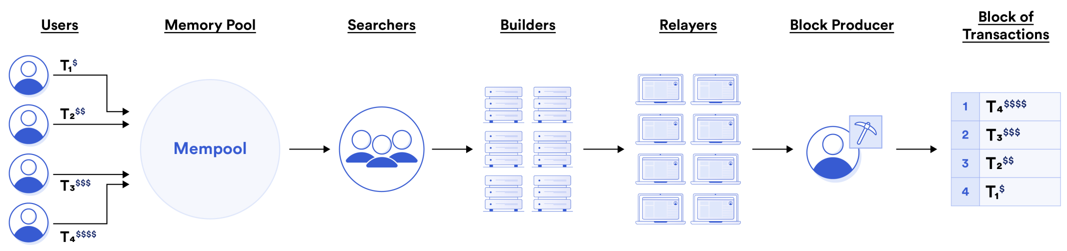


Figure 3.1: MEV by third-party networks [40]

As represented in Figure 3.1, the extraction of MEV often involves outsourcing block creation to third-party networks of searchers, builders, and relayers. Searchers identify MEV opportunities and create bundles of transactions, which builders then combine into full block payloads. Relayers connect these full blocks to the blockchain's block producers. This is just one example of how MEV is extracted, and the ecosystem continues to evolve rapidly.

Flash Loans

Flash loans allow users to borrow large amounts of cryptocurrency without collateral. The atomicity of blockchain transactions ensures that the loan is borrowed at the beginning of a transaction and repaid with a fee by the end of it. Even though they are not inherently harmful themselves, they can facilitate sophisticated attacks like price oracle manipulation, where an attacker temporarily inflates the value of an asset to exploit price discrepancies. Since flash loans have no real-world counterpart, understanding blockchain state management within blocks is essential.

3.2 Smart Contract Vulnerabilities

SCs form the backbone of Web3 technology. When a user interacts with a decentralized application, they are most likely engaging with a trustless, decentralized, and transparent SC. Given their immutable and decentralized nature, ensuring the contract's security prior to deployment is crucial.

In the context of blockchain, vulnerabilities are flaws or weaknesses in the design, implementation, or use of Blockchain Technology that can be exploited to perform malicious or unwanted actions. These vulnerabilities can exist in the code of SCs and can lead to significant losses, as demonstrated by numerous attacks, such as the one against Euler Finance [6]. That exploit, fuelled by vulnerabilities within the protocol's SCs, allowed attackers to orchestrate a sequence of transactions that culminated in a massive embezzlement of $197 million.

In the following sections, we will explore these vulnerabilities in detail, examining their nature, potential impacts, and the strategies that can be employed to mitigate them.

3.2.1 Reentrancy

Reentrancy [41], also known as a recursive call attack, is one of the most critical vulnerabilities to address when implementing SCs. Reentrancy occurs when contract A calls contract B, and B then calls back into A, potentially executing A's code again. This situation arises due to Solidity's fallback mechanism. When the caller uses the `call` low-level function without providing a matching function signature, the callee's `fallback` function is triggered. This function can then call the caller's function, leading to re-entrance into the caller. Under certain circumstances, this mechanism can cause unexpected and uncontrolled transfers of Ether.

For example, consider the scenario illustrated in Figure 3.2: An attacker calls the `withdraw` function in `Victim Contract` to request a withdrawal of their balance. The function uses the `call` low-level function to send Ether. The fallback function of the `Attacker Contract` responds to this call by invoking `withdraw` function again. This second call is nested within the initial `withdraw` call, as the first one has not yet completed. This recursive call allows the attacker to exploit the outdated status of their balance.

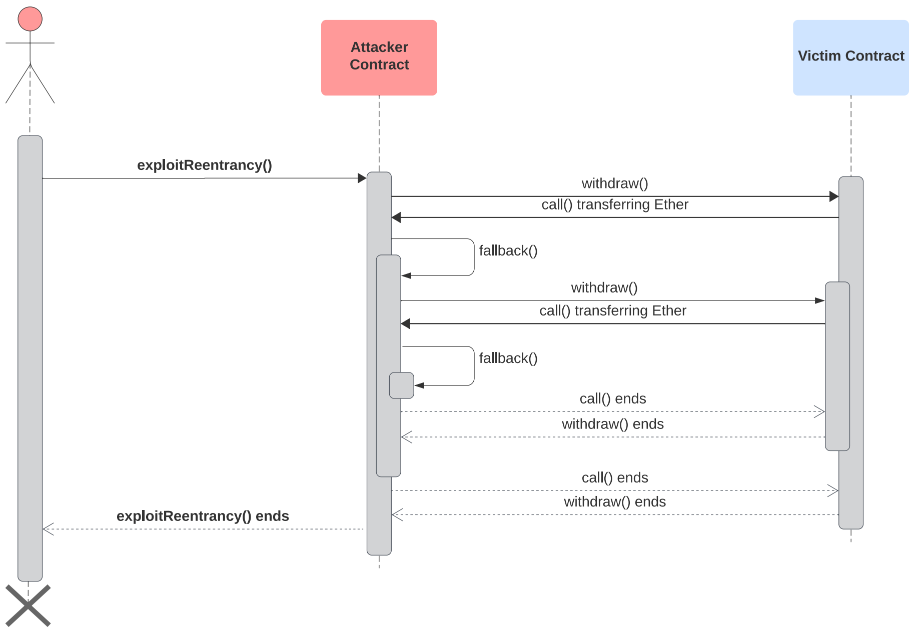


Figure 3.2: Reentrancy Exploit

While the described reentrancy scenario is the most basic and, thus, easier to spot, there are other more complex forms of reentrancy that significantly increase the likelihood of them going unnoticed. One such form is read-only reentrancy, where an attacker manipulates the read-only state of a contract to influence its behavior unexpectedly. Another form is cross-function reentrancy, which involves exploiting interactions between multiple functions within the same contract, making it more challenging to detect and mitigate. One more is cross-contract reentrancy, which exploits the interactions between multiple contracts, creating loops that are harder to trace and control. And even more, as blockchain ecosystems become more interconnected, cross-chain reentrancy also poses a significant risk.

The checks-effects-interactions pattern [42] is a reliable approach to prevent reentrancy attacks. This pattern defines how a function's code should be organized to minimize undesirable side effects and execution behaviors. The function should first include all necessary checks (e.g., `assert()` and `require()` statements). If these checks pass, the function should resolve all the contract's effects (e.g., updating balances). Only after all state changes have been made should the function interact with other contracts. This approach ensures that even if an attacker performs a recursive call, they cannot exploit the contract's state, as external interactions are handled last.

3.2.2 Transaction Ordering Dependency

Transaction ordering dependency, also known as *race conditions*, occurs when the execution logic of a SC depends on the order of submitted transactions. In Ethereum, miners (or validators) determine the order of transactions within a block. Since transactions are visible to the network before being executed, participants can exploit this visibility by sending transactions with a higher gas price to ensure their transactions are processed first.

As discussed in the previous Chapter, this is called MEV, and is often exploited in different ways. The most common strategy is *sandwich-attacks*, as depicted in Figure 3.3. For instance, Searchers often buy tokens at lower prices just before a significant transaction increases their value, and then sell the tokens to profit at the expense of the victim transaction. This exploitation can lead to significant financial losses and undermine the fairness of the blockchain ecosystem.

To protect against *front-running* and transaction ordering dependencies, it's essential to eliminate the advantage gained from transaction ordering. One effective solution is to use a commit-reveal scheme. In this scheme, participants initially submit the hash of their response rather than the response itself. The contract stores this hash along with the sender's address. Once all responses are submitted, participants reveal their actual responses. This approach ensures that no participant can gain an unfair advantage based on transaction ordering, as the actual data remains hidden until all commitments are made.

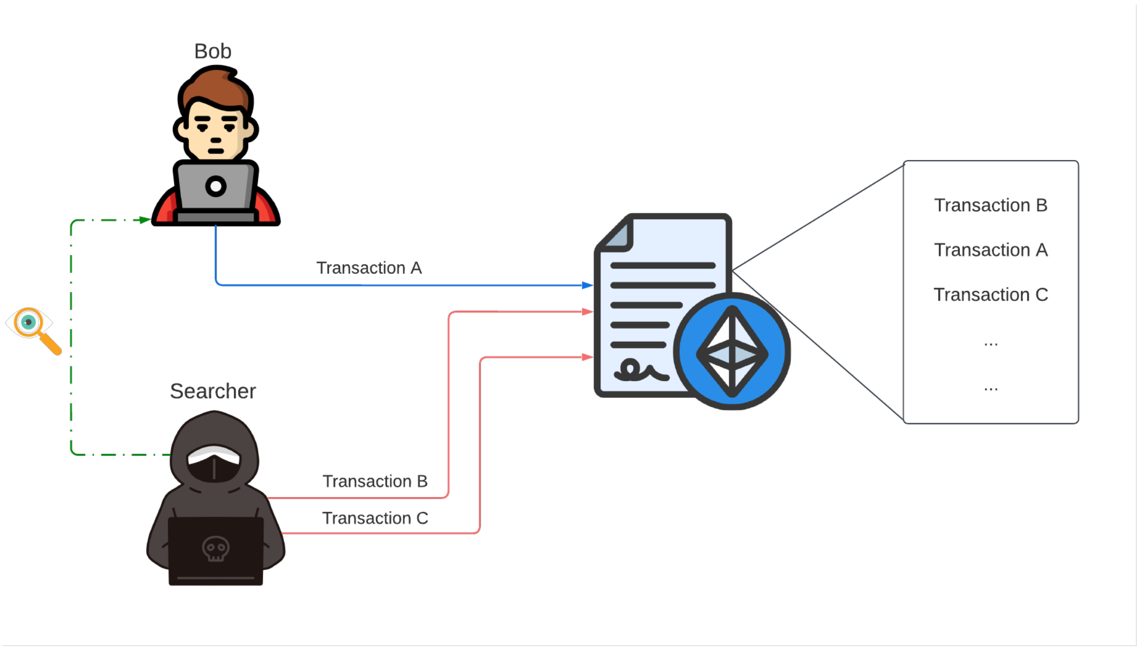


Figure 3.3: Sandwich Attack

3.2.3 Arithmetic Vulnerabilities

Arithmetic vulnerabilities in Ethereum SCs primarily revolve around two key issues: integer overflow and underflow, and precision loss. Both can lead to unpredictable contract behavior, significant security breaches, or financial losses.

Integer Overflow and Underflow

Solidity's integer types, such as uint8, uint16, and uint256, have fixed sizes, which means they can represent numbers only within specific ranges. Arithmetic operations that result in values outside these permissible ranges cause integer overflow or underflow. For example, if a uint8 type (which can hold values from 0 to 255) is incremented beyond 255, it will wrap around to 0, causing an overflow. Similarly, decrementing below 0 results in underflow, wrapping around to 255.

Solidity 0.8.0 introduced built-in overflow and underflow checks that automatically revert transactions when such conditions are detected. However, developers should remain vigilant, especially when using inline assembly code. Inline assembly allows for more granular control but bypasses Solidity's built-in safety checks, making manual validation crucial.

Precision Loss

The EVM does not natively support floating-point arithmetic, a design decision rooted in determinism. Instead of relaying on floating points, developers work within the confines of fixed-points arithmetic. In this system, numbers are represented as integers, and developers must use them to simulate floating-point behavior. This approach is common in financial applications, such as the ERC20 token standard, which uses a `decimals` field to represent the number of digits after the decimal point.

Precision loss occurs because Solidity's division operation always rounds down to the nearest integer. This truncation can lead to inaccuracies, particularly in applications requiring high precision. For example, in a contract managing fractional tokens, rounding errors can accumulate, resulting in significant discrepancies over time.

3.2.4 Mishandled Exceptions

Solidity offers two primary paradigms for interacting with external contracts: direct contract calls and low-level calls.

Direct contract calls involve invoking functions directly on known contract interfaces. When an exception occurs during these calls, the transaction is reverted, and gas is consumed. This automatic reversion helps maintain contract integrity by ensuring that failed transactions do not produce unintended effects.

Conversely, low-level calls are executed using methods like `call()`, `delegatecall()`, or `staticcall()`. These methods return a boolean success flag instead of reverting the transaction on exceptions [43]. Unlike direct calls, low-level calls do not automatically revert transaction execution upon failure. Instead, they return values that need to be checked to ensure the intended execution flow. Failure to adequately check the result of a low-level call can lead to unintended continuation of execution, potentially compromising contract logic and security. Proper handling and validation of these return values are crucial to maintaining the security and correctness of the contract.

3.2.5 Delegatecall to Insecure Contracts

The DELEGATECALL opcode is a unique method that enables a contract to execute code from another contract while maintaining its own context [44]. Unlike a conventional message call, where the execution context is that of the target contract, DELEGATECALL executes the code at the target address but within the context of the calling contract. This means the calling contract's address, storage, and balance remain unchanged.

Delegatecall allows a SC to dynamically load and execute code from another contract at runtime. However, this also means that the called contract can modify the caller's state variables because the delegatecall function preserves the execution context. This capability introduces significant risks, especially when calling untrusted contracts via `delegatecall()`. Since the code at the target contract has full control over the caller's balance and storage, it can potentially alter any of the caller's storage data.

To mitigate these risks, it is crucial to avoid making `delegatecall()` to addresses derived from user inputs. Ensuring that it is only used with trusted contracts can help prevent malicious manipulation of contract state and enhance overall security.

3.2.6 Dependence on this.balance

Depending on `this.balance` in Solidity contracts introduces significant vulnerabilities. It can be exploited in various ways, making it an unreliable source for managing contract logic or conditions. Two primary methods of exploiting this vulnerability include using the SELFDESTRUCT opcode [44] and pre-calculating the contract address to send Ether before deployment.

First, the SELFDESTRUCT operation can forcefully send Ether to any contract, regardless of its design or intention. When a contract is destroyed using SELFDESTRUCT, its remaining Ether is sent to a specified address. Malicious actors can exploit this behavior to manipulate the balance of a target contract. If a contract's logic relies on `this.balance` for conditions or operations, an attacker can disrupt its functioning by artificially inflating its balance through SELFDESTRUCT.

Second, attackers can pre-calculate the address where a contract will be deployed and send Ether to that address beforehand. When the contract is deployed, it will start with a balance that was not intended by the contract creator. This initial balance can affect the contract's logic, if the contract uses `this.balance` to determine certain actions. This exploit leverages the predictability of Ethereum contract addresses, allowing attackers to manipulate contract behavior from the outset.

To mitigate these risks, developers should avoid relying on `this.balance` for critical contract logic. Instead, they should use internal accounting mechanisms to track balances.

3.2.7 Weak Pseudorandom Number Generator

Generating truly random numbers in blockchain systems is inherently impossible due to their deterministic nature. A weak source of randomness in a SC allows attackers to predict or manipulate outcomes. SCs often rely on random values for critical decisions, such as selecting lottery winners or assigning tasks in decentralized systems. If the randomness source is predictable, an attacker can exploit this, undermining the contract's integrity.

Blockchain-based applications frequently leverages certain secret seeds to mimic randomness. However, because everything on the blockchain is visible to all participants, seeds cannot be privately stored in a SC. Current practices typically involve using block-related information such as `block.timestamp` or `block.hash`. Which is inherently insecure, as it can be anticipated and slightly modified by miners [45].

To address these vulnerabilities, adopting more reliable sources of randomness is essential. One robust solution is employing oracles [46] that provide verified randomness. Figure 3.4 illustrate Chainlink's Verifiable Random Function (VRF) [47], which offers a secure method for generating randomness that cannot be manipulated by miners or other participants. Developers can implement this by inheriting from the `VRFConsumerBase` [48] contract provided by Chainlink [49]. This approach ensures the randomness used is both unpredictable and resistant to manipulation.

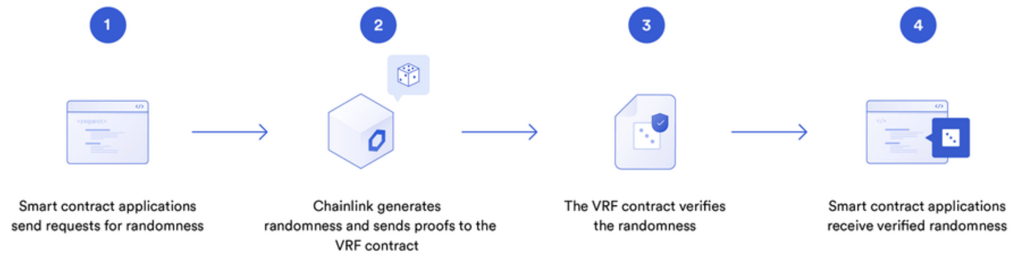

1
Smart contract applications send requests for randomness
2
Chainlink generates randomness and sends proofs to the VRF contract
3
The VRF contract verifies the randomness
4
Smart contract applications receive verified randomness

Figure 3.4: Chainlink's Verifiable Random Function [47]

Alternatively, cryptographic commitment schemes, such as RANDOA [50], provide another effective approach. These schemes involve participants committing to their inputs in a concealed manner, which are later revealed collectively to generate a random outcome. This method ensures that no single participant can influence the result based on others' commitments, thus enhancing fairness and security.

3.2.8 Transaction Origin Use

The `tx.origin` [51] variable in Solidity is a unique global variable that stores the address of the original caller of a transaction, unlike `msg.sender`, which refers to the immediate caller. Using `tx.origin` as an authorization parameter can expose a SC to significant security risks. For example, if an authorization check such as `require (tx.origin == owner);` is implemented, an attacker's contract could take advantage of this by being called by the real owner and then calling the vulnerable contract, as depicted in Figure 3.5, thus passing the check and executing malicious code.

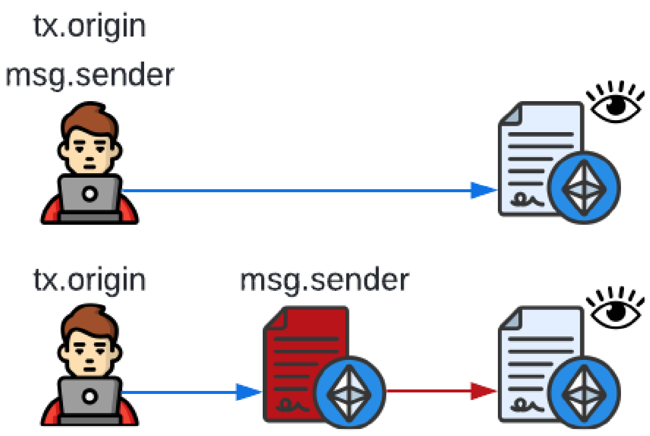


Figure 3.5: tx.origin and msg.sender

This vulnerability arises because attackers can execute their code using the caller's `tx.origin`, effectively impersonating the original sender. Therefore, `tx.origin` should never be used for identity verification or authorization purposes.

By using `msg.sender` instead, developers can ensure that the authorization checks are performed against the immediate caller, thereby preventing such exploits and enhancing the security of the SC.

3.2.9 Default Visibility

Solidity provides visibility specifiers—public, private, external, and internal—that control the accessibility of functions and variables from outside the contract. By default, if a developer does not specify the visibility of a function or variable, it is set to public. This means that if an internally-used function is not explicitly marked as private or internal, it can be called from outside the contract, potentially leading to unexpected operations and security vulnerabilities [51].

To mitigate this risk, it is essential to consistently define the visibility of all functions and variables in a contract. Even functions that are intended to be public should have their visibility explicitly specified. This practice helps ensure that the contract's behavior is predictable and secure, preventing unauthorized access and operation escalation.

3.2.10 Denial of Service by an External Call

When control is transferred to an external contract, the caller contract's execution can fail accidentally or deliberately, leading to a Denial of Service (DoS) state. This can occur when a transaction is reverted due to a failure in the external call, or when the callee contract deliberately causes the transaction to revert, disrupting the caller contract's execution.

To mitigate this vulnerability, it is advisable to place any external call initiated by a contract into a separate transaction. By doing so, the impact of a failed external call is isolated, preventing it from causing a DoS state in the caller contract. Additionally, adopting a "pull" pattern—where recipients withdraw funds themselves rather than having the sender "push" funds to them—can help minimize this risk.

3.2.11 Denial of Service with Block Gas Limit

Each block in the Ethereum blockchain has a gas limit, which is the maximum amount of gas that can be spent within that block [52]. If a transaction exceeds this limit, it fails with an *Out Of Gas* exception, resulting in a DoS. A common scenario for this issue involves dynamic data structures, such as arrays that grow over time. As these arrays increase in size, the gas required to process them can exceed the block gas limit, causing transactions that interact with the array to fail.

To prevent this type of DoS, developers should design contracts with efficient data structures and consider the potential growth of dynamic data. Employing techniques like batching operations or limiting the size of data structures can help manage gas consumption and avoid exceeding the block gas limit.

3.2.12 Price Oracle Manipulation

In blockchain, *incorrect oracle usage* refers to situations where an oracle is misused, leading to the injection of inaccurate or malicious data. An oracle is a trusted third-party service that supplies external data to a blockchain-based SC. Inappropriate use of oracles can result in a variety of issues, including improper execution of SCs, security flaws, and financial losses for users.

Determining the price of an asset is a critical functionality for many business models. In DeFi, this is typically accomplished using price oracles. One of the most prevalent types of price oracles is the Automated Market Maker (AMM), designed for exchanging two types of assets, such as WETH and USDC. Users can exchange one asset for another, and the exchange rate is determined by a predefined invariant law. In Uniswap [53], a leading AMM, this invariant is represented by the constant product formula, $x \cdot y = k$, where x and y are the reserves of the two assets and k is a constant. The price of one asset relative to the other is determined by their ratio, y/x , indicating the price of WETH in terms of USDC. An increased supply of x leads to its depreciation and $y's$ appreciation.

Although price oracles are crucial for DeFi project development, they are sometimes misused by application contracts, making their price queries vulnerable. This issue is not due to a bug in the price oracle contract but rather due to improper implementation in the application contract. For instance, although Uniswap provides an official and well-protected API for price queries, developers often implement their own queries to avoid the high gas costs associated with the official API. A common faulty code pattern involves determining the price by querying the ratio of an asset's instant balances in the oracle contract.

Due to the atomic nature of blockchain transactions, a malicious user can manipulate the price without interruption. Moreover, they can make use of flash loans to do so. As discussed previously, Flash loans represents a unique and innovative lending model enabled by Blockchain Technology, allowing users to borrow large amounts of funds without collateral. The atomicity of blockchain transactions ensures that the loan is borrowed at the beginning of a transaction and repaid with a fee by the end of it. Flash loans can facilitate sophisticated attacks that exploit the interconnectedness of DeFi, even though they are not inherently harmful themselves. Since flash loans have no real-world counterpart, understanding blockchain state management within blocks is essential.

Figure 3.6 presents an example of a flash loan attack on a DeFi lending protocol using a two-asset ratio query as a price oracle. The attacker borrows a large amount of token A from a protocol that supports flash loans. They then swap

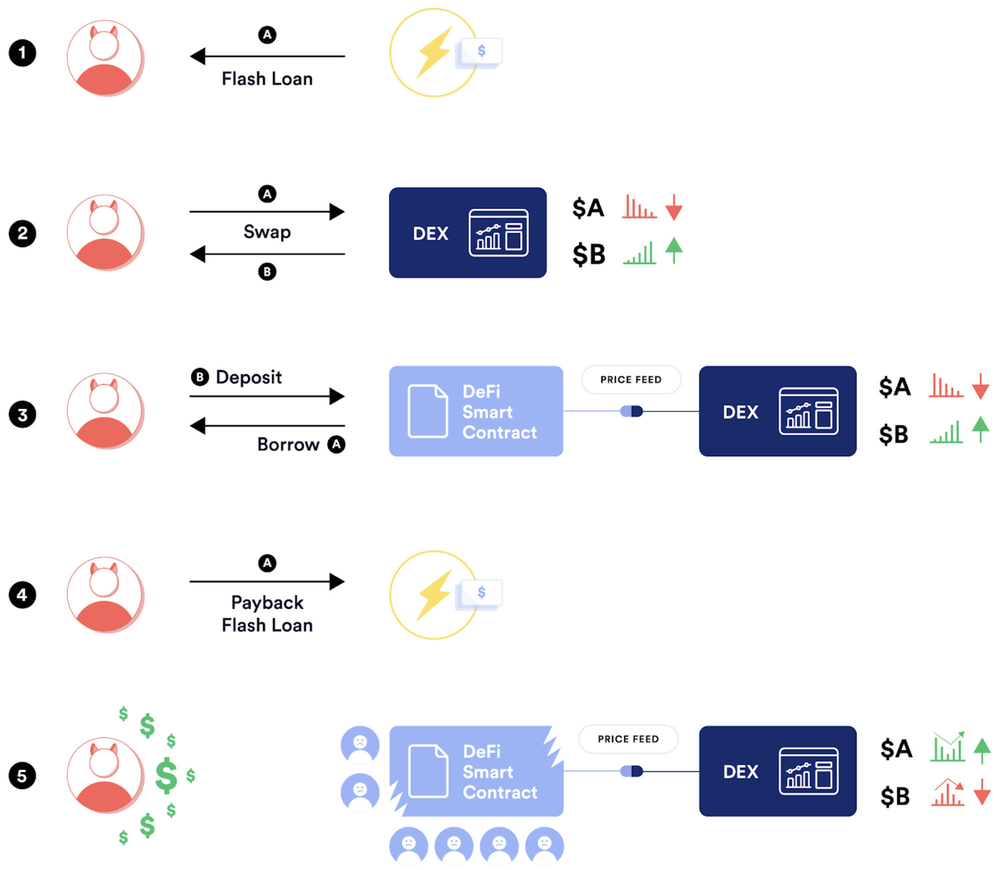


Figure 3.6: Steps taken by a malicious actor during a flash loan price oracle attack [54]

token A for token B on a Decentralized Exchange (DEX)[1], which lowers the spot price of token A and increases the spot price of token B on the DEX. Next, the attacker deposits the purchased token B as collateral on a DeFi protocol that uses the spot price from the DEX as its sole price feed. By using the manipulated spot price, they borrow a larger amount of token A than should normally be possible. The attacker then uses a portion of the borrowed token A to fully pay back the original flash loan, keeping the remaining tokens and generating a profit from the manipulated price feed. As the spot prices of tokens A and B on the DEX are arbitraged back to the true market-wide price, the DeFi protocol is left with an

[1]A DEX is a peer-to-peer marketplace where transactions occur directly between cryptocurrency traders without the need for an intermediary.

undercollateralized position.

To mitigate the risks of price oracle manipulation, it is crucial to ensure that exchange rates are obtained from reliable sources and are not susceptible to sudden fluctuations. This can be achieved by using secure and well-designed oracle services, incorporating proper checks, and avoiding reliance on instant balance queries that can be manipulated using flash loans [51].

3.2.13 ID Uniqueness Violation

Most SC functionalities involve entities operating on assets. Within SC implementations, these entities and assets are typically represented as data structures with an ID field that uniquely identifies them. However, developers may sometimes neglect to ensure the uniqueness of these ID fields, mistakenly assuming other data fields are unique and using them as replacement IDs. This oversight can lead to security vulnerabilities where an adversary impersonates an entity or creates a fake or duplicate asset with the same field value as a legitimate entity or asset. This allows the adversary to bypass access control checks and perform unauthorized operations.

Ensuring the uniqueness of ID fields is crucial to maintaining the integrity and security of SCs. Developers should implement strict checks to guarantee that each ID is unique and cannot be duplicated or spoofed. This can involve using cryptographic techniques or decentralized identity solutions to generate and verify unique IDs for all entities and assets within the SC

3.2.14 Inconsistent State Updates

SCs have many state variables with implicit correlations. However, when developers update one variable, they may forget to update the correlate variable(s) or update incorrectly. Depending on the state variables that are incorrectly updated, the consequences of this kind of bugs range from incorrect statics to loss of funds.

To avoid this vulnerability, developers should use diagrams to visually represent the relationships between state variables. This helps ensure that all correlations are clearly understood and maintained throughout the contract's lifecycle, reducing the risk of inconsistencies.

3.2.15 Improper Access Control

Access control regulates the authority of individuals within a contract to perform specific actions, typically essential for the protocol's safety and stability. Its significance lies in preventing malicious parties from abusing vital contract functions. This is achieved by granting activation privileges for specific operations to designated accounts and implementing safeguards to prevent unauthorized parties from initiating restricted operations.

However, vulnerabilities can arise when there is an unexpected sequence of functions that can be invoked to bypass these access controls, leading to privilege escalation.

Privilege escalation bugs occur when an attacker can exploit an unexpected path to gain unauthorized access to sensitive operations. Identifying these vulnerabilities is challenging because it requires recognizing sensitive operations, which often demands domain-specific knowledge, and mapping out all possible paths that could lead to these operations.

Proper implementation of access control involves defining clear roles and permissions within the contract, ensuring that only authorized accounts can execute sensitive functions. Additionally, robust auditing and monitoring mechanisms should be in place to detect and respond to any unauthorized access attempts promptly.

3.2.16 Atomicity Violations

In SCs, multiple business flows can interleave and interfere with each other by accessing the same state variables. Some business flows require business level atomicity, meaning state variables must not be accessed by other flows while they are ongoing. Developers often fail to anticipate such interference and neglect to ensure atomicity, mistakenly believing that atomicity is guaranteed by the runtime. However, the runtime only guarantees the atomicity of individual transactions, not the atomicity of entire business flows. Ensuring business flow atomicity, if needed, is the responsibility of the developers.

3.3 Detection Methods

Given the potential for significant financial losses due to vulnerabilities in SCs, various methods and tools have been proposed to identify vulnerabilities in Solidity SCs [55]. The most common approaches include static analysis, dynamic analysis, and formal verification.

Static Analysis

Static analysis involves debugging by reviewing source code before it is run, comparing the code against predefined coding rules. Techniques for static analysis include:

- **Taint Analysis**: Identifies variables that may be tainted by user inputs.
- **Control Flow Graph Analysis**: Examines the flow of control within the code.
- **Symbolic Analysis**: Uses symbolic values to represent different possible inputs and states.

Dynamic Analysis

Dynamic analysis involves observing code execution in its original context. It simulates an attacker feeding malicious or unpredictable inputs to functions to identify vulnerabilities. The primary technique is:

- **Fuzzing**: Automated testing that injects incorrect, malformed, or unpredictable inputs to uncover flaws and vulnerabilities. The tool monitors for issues such as crashes or data leaks.

Formal Verification

Formal verification automates bug detection by comparing a formal system model to formal requirements or behavior specifications. Through mathematical analysis, it provides a high level of confidence in the system's correctness.

3.4 Vulnerability Detection Tools

Various tools are available to test SCs and identify vulnerabilities. The most popular ones include Oyente, Slither, Mythril, Manticore, and Echidna.

Oyente

Oyente [56] is the first symbolic execution tool for Ethereum SCs, detecting seven types of vulnerabilities: Reentrancy, Integer overflow/underflow, Transaction order dependence, Timestamp dependence, Callstack Depth, EVM Code Coverage, and Parity Multisig bugs.

Slither

Slither [57] is an open-source static analysis tool known for its efficiency in auditing SCs. It is user-friendly, with installation options via Docker or Python package managers. Slither detects approximately 70 types of vulnerabilities, enhances code understanding through visual aids like contract graphs, and checks SC compliance with Ethereum standards such as ERC-20 and ERC-777.

Mythril

Mythril [58] uses symbolic execution to identify potential weaknesses in SCs. It generates a Control Flow Graph and performs symbolic execution of EVM bytecode to narrow down the search area for vulnerabilities.

Manticore

Manticore [59] detects vulnerabilities in SCs using symbolic execution to explore distinct computation paths in EVM bytecode.

Echidna

Echidna [60] is a property-based fuzzing tool for SCs. It focuses on violating user-defined invariants that represent potential contract errors rather than just finding crashes.

Empirical studies have evaluated the effectiveness of this tools [61] [62]. This study [7] systematically investigate 516 unique real-world SC vulnerabilities in years 2021-2022, and study how many can be exploited by malicious users and cannot be detected by existing analysis tools. Eventually, they find that more than 80% exploitable bugs are beyond existing tools. Therefore, despite the availability of these tools, developers still rely heavily on manual vulnerability detection [63]. Thus, providing them with security smells and vulnerability mitigation skills is crucial. Underpinning our objective to contribute to the improvement of auditors' and developers' awareness of security issues.

Part III

Project Development

Chapter 4

Faillapop Project

Content

Synopsis

This chapter provides a comprehensive examination of the Faillapop project, a decentralized, vulnerable-by-design protocol developed to enhance educational experiences in Ethereum security. The project was initiated by José Carlos Ramírez

Vega, with whom I have been collaborating, with the goal of bridging the gap between simple CTF exercises and the complexities of real-world blockchain protocols.

We begin by exploring the origins and objectives of Faillapop, highlighting its architecture, which involves multiple interdependent SCs. This setup provides a realistic environment for students and professionals to test and hone their skills in securing blockchain applications.

An initial insight into the most popular CTFs, along with a study in common vulnerabilities in public contests and exploited protocols, underscores the need for a more sophisticated approach to security education. This need is precisely what Faillapop aims to address.

4.1 Origin and Objectives of Faillapop

Faillapop is a vulnerable buy-and-sell protocol conceived out of the necessity to create a more comprehensive and realistic training environment for Ethereum security enthusiasts. The project's inception can be traced back to the educational efforts of José Carlos Ramírez Vega, who identified significant gaps in existing educational tools while teaching Blockchain Technology at the University of Málaga.

José Carlos, a telematics engineer and blockchain security auditor by profession, also lectures on Solidity Security and Auditing in the Extension Course on Blockchain Technologies at the University of Málaga. In 2023, he created a GitHub repository containing a variety of practical examples and exercises for his lectures. However, he wanted to go one step further and came up with Faillapop.

Common CTF exercises, while beneficial, often focus on isolated vulnerabilities within single SCs. These exercises, although complex, do not fully represent the multifaceted nature of real-world blockchain applications, which typically involve multiple SCs interacting with one another. This create a more intricate web of dependencies and potential vulnerabilities, increasing the likelihood of them being overlooked. This complexity is precisely what Faillapop aims to replicate.

Faillapop was designed to bridge this educational gap by simulating a fully functional decentralized application composed of various interdependent SCs. This approach not only makes the training more challenging but also more reflective of actual conditions encountered in professional blockchain development and security auditing. By working within this realistic framework, students and professionals can develop a deeper understanding of how vulnerabilities can emerge and propagate within a multi-contract environment.

Additionally, Faillapop serves a dual purpose by acting as a testing ground for companies seeking to evaluate the skills of potential hires or to train their current developers and auditors. By integrating Faillapop into their training regimes, educational institutions and companies can ensure that their learners and employees are better prepared for the complexities of blockchain security. This alignment with practical needs makes Faillapop a crucial tool in the advancement of blockchain education and security practices.

Through this collaborative effort, Faillapop is positioned as a pioneering educational tool that combines theoretical knowledge with practical application, thereby addressing the limitations of existing CTF exercises and contributing significantly to the field of Ethereum SC security.

4.1.1 Initial Protocol Architecture

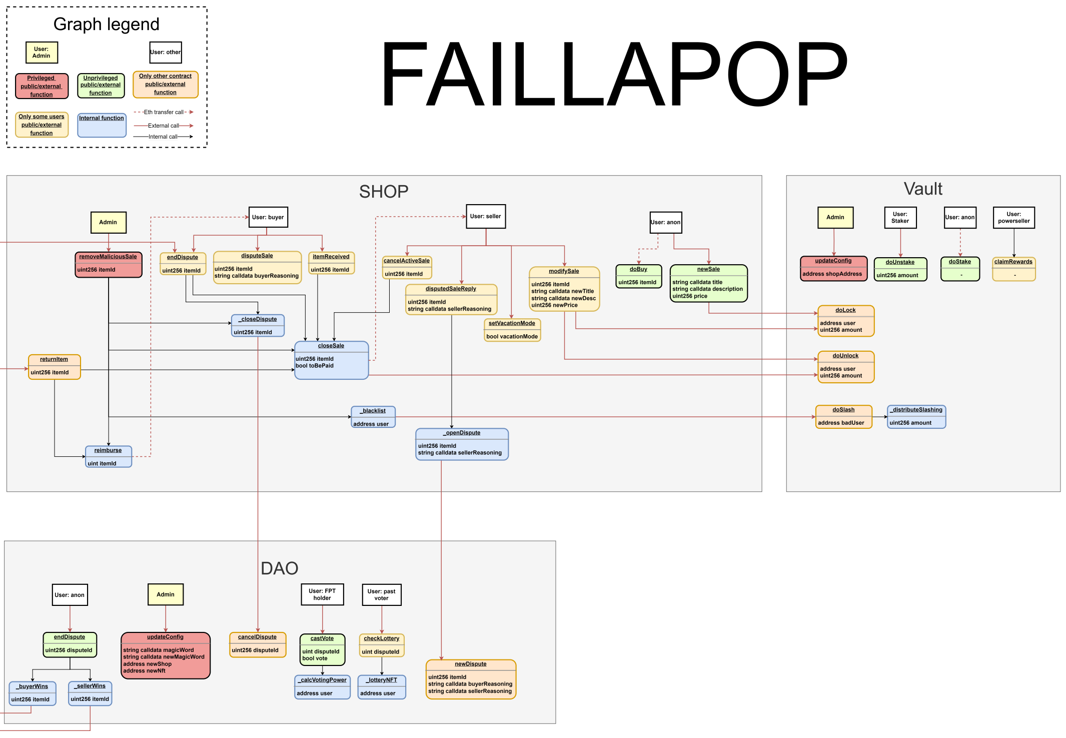

Graph legend
User: Admin
User: other
Privileged public/external function
Unprivileged public/external function
Only other contract public/external function
Only some users public/external function
Internal function
Eth transfer call
External call
Internal call
FAILLAPOP
SHOP
Admin
User: buyer
User: seller
User: anon
removeMaliciousSale uint256 itemId
endDispute uint256 itemId
disputeSale uint256 itemId string calldata buyerReasoning
itemReceived uint256 itemId
cancelActiveSale uint256 itemId
disputedSaleReply uint256 itemId string calldata sellerReasoning
modifySale uint256 itemId string calldata newTitle string calldata newDesc uint256 newPrice
doBuy uint256 itemId
newSale string calldata title string calldata description uint256 price
setVacationMode bool vacationMode
_closeDispute uint256 itemId
closeSale uint256 itemId bool toBePaid
returnItem uint256 itemId
_blacklist address user
_openDispute uint256 itemId string calldata sellerReasoning
reimburse uint itemId
Vault
Admin
User: Staker
User: anon
User: powerseller
updateConfig address shopAddress
doUnstake uint256 amount
doStake -
claimRewards -
doLock address user uint256 amount
doUnlock address user uint256 amount
doSlash address badUser
_distributeSlashing uint256 amount
DAO
User: anon
Admin
User: FPT holder
User: past voter
endDispute uint256 disputeId
updateConfig string calldata magicWord string calldata newMagicWord address newShop address newNft
cancelDispute uint256 disputeId
castVote uint disputeId bool vote
checkLottery uint disputeId
_calcVotingPower address user
_lotteryNFT address user
newDispute uint256 itemId string calldata buyerReasoning string calldata sellerReasoning
_buyerWins uint256 itemId
_sellerWins uint256 itemId

Figure 4.1: Faillapop's initial architecture

Figure 4.1 shows the initial architecture of Faillapop [64], before my collaboration, consisted of three SCs—Shop, DAO, and Vault—that interact with each other to create a decentralized buying and selling platform. These three SCs collectively establish the initial framework of Faillapop. Each contract plays a crucial role in creating a decentralized, transparent, and secure marketplace for users to buy and sell goods. This architecture sets the stage for further contributions, aimed at improving the platform's functionality and its closeness to real-world applications.

Shop Contract

The Shop contract is the core component where users interact to post sales, make purchases, and open disputes. Users can create new sales by providing details such as the title, description, and price of the item. Buyers can then purchase items, and if they encounter issues, they have the option to open a dispute. The Shop contract manages the state of each sale and dispute, ensuring a smooth transaction process. Key functionalities include:

- **New Sale**: Allows sellers to post items for sale.
- **Buy Item**: Enables buyers to purchase items by sending the exact amount of Ether.
- **Open Dispute**: Provides buyers a way to raise disputes if they are unsatisfied with their purchase.
- **Resolve Sale**: Allows confirmation of receipt or closing of disputes based on DAO decisions.
- **Modify and Cancel Sale**: Enables sellers to update or cancel their listings.

DAO Contract

The DAO contract is responsible for handling disputes raised by buyers through a decentralized voting mechanism. Members of the DAO can vote on the disputes to determine their outcome. This ensures that the resolution process is fair and transparent, leveraging the collective decision-making power of the community. Key functionalities include:

- **Create Dispute**: Initiates a dispute based on buyer and seller reasoning.
- **Vote on Dispute**: Allows DAO members to cast their votes either in favor of the buyer or the seller.

- **Resolve Dispute**: Finalizes the outcome of a dispute based on the votes received.
- **Award NFTs**: Incentivizes voters with NFTs for participating in dispute resolutions.

Vault Contract

The Vault contract acts as the financial backbone of the platform, storing and managing the deposits made by sellers. To list an item for sale, sellers must lock funds in the Vault, which serves as a security deposit to discourage malicious behavior. In case of disputes, these funds can be slashed or returned based on the DAO's decision. Key functionalities include:

- **Stake Funds**: Allows users to deposit Ether into the Vault.
- **Lock and Unlock Funds**: Manages the locking of funds for active sales and their release upon completion or dispute resolution.
- **Slash Funds**: Penalizes malicious sellers by slashing their staked funds.
- **Claim Rewards**: Enables users to claim rewards generated from slashed funds.

The development and testing of Faillapop have utilized the Foundry framework, which has significantly contributed to the project's robustness and versatility. Foundry is a comprehensive Ethereum development toolkit designed to streamline the process of writing, testing, and deploying SCs. It includes a variety of tools that are essential for blockchain development.

Foundry's key components include:

- **Forge**: A fast, modular, and extendable command-line tool for Ethereum development. Forge allows developers to compile, deploy, and interact with SCs seamlessly. Its speed and efficiency make it ideal for iterative development and testing cycles.
- **Cast**: A powerful command-line tool for interacting with Ethereum contracts and sending transactions. Cast enables developers to perform complex operations and queries directly from the terminal, enhancing the development workflow.

- **Anvil**: An Ethereum RPC endpoint for local testing, providing a fast and lightweight environment to simulate blockchain interactions. Anvil's integration with Forge ensures a smooth development experience, from initial coding to final deployment.

The Foundry framework's emphasis on speed, modularity, and ease of use aligns perfectly with the objectives of Faillapop. By leveraging Foundry, we were able to create a realistic and dynamic environment for testing SC security. This framework not only facilitated the development of Faillapop but also ensured that the educational exercises were both challenging and reflective of real-world scenarios.

4.2 Initial Security Evaluation

The initial phase of my contribution to the Faillapop project consisted of a comprehensive security analysis to establish a solid understanding of both the initial state of the protocol and the current context of actual educational tools and protocols in terms of security. This phase was critical for identifying potential areas for improvement. The security analysis consisted of several key activities: an audit of the protocol, participation in CTF exercises, and a study of frequent vulnerabilities in public contests and exploited protocols. These activities provided valuable insights into the security of Ethereum SCs, forming a solid foundation for the further development and improvement of Faillapop.

4.2.1 Initial Project Audit

I conducted an audit of the initial state of the protocol, involving a thorough review of the SCs' codebase to identify security flaws, logic errors, and potential points of exploitation. The audit employed both automated static tools and manual inspection to ensure comprehensive coverage. The results of this audit guided the direction of future improvements.

Audit Methodology

Initially, the SCs were not tested, so the audit was based on a thorough code review. The audit methodology employed was a multi-step process, combining systematic approaches with in-depth analysis:

1. **Code Structure Analysis**:

- **Objective**: Understand the overall architecture of the SCs.
- **Approach**: Examining the declared variables, utilized functions, and overall code layout. This step provides an initial understanding of how the contract is designed and its intended functionality.

2. **Threat Modeling**:
    - **Objective**: Anticipate potential security threats and identify critical points that need careful scrutiny.
    - **Approach**: Consider each execution flow, possible pitfalls, and exploitation points from an attacker's perspective. This involves thinking like an attacker to foresee where and how they might attempt to exploit the contract.
3. **Initial Code Reading**:
    - **Objective**: Gain a general understanding of the implementation.
    - **Approach**: Conduct a light reading of the code to get a broad sense of its functionality, followed by detailed readings to fully understand specific sections. This phase helps identify obvious issues and informs the areas that need more focused analysis.
4. **Detailed Vulnerability Identification**:
    - **Objective**: Systematically identify all potential security issues.
    - **Approach**: Conduct a thorough analysis of each part of the code, cross-referencing with known vulnerability patterns, and using both automated tools (such as static analysis tools) and manual inspection. This step includes updating the list of possible threats based on the detailed analysis.
5. **Execution Flow Analysis**:
    - **Objective**: Ensure that the contract's logic is secure throughout its execution.
    - **Approach**: Analyze each execution flow in detail to detect potential issues, ensuring that each function operates as intended and cannot be exploited.
6. **Proof of Concept**:
    - **Objective**: Validate the identified vulnerabilities.

- **Approach**: Implement and demonstrate proof of concept attacks for each identified vulnerability to confirm their existence and understand their impact.

The audit of the initial state of the Faillapop protocol revealed several critical vulnerabilities and provided deep insights into the protocol's design and implementation. These insights were invaluable for proposing design improvements and planning future implementations.

4.2.2 Engagement in Capture the Flag Exercises

I then engaged in various CTF exercises, which are effective in teaching and testing security skills. These exercises simulate real-world hacking scenarios, providing valuable insights into the tactics and techniques used to exploit SCs.

We decided that the most interesting exercises, due to their popularity, were the following:

- **Ethernaut [65]**: Developed by Open Zeppelin [66], one of the leading companies in the field of SCs security. This company specialises in the development of libraries and tools for the creation of secure SCs on different blockchain platforms, such as Ethereum. The proposed CTF is a composition of 27 challenges in areas such as access control, payment management, token handling and other critical functionalities in the development of secure contracts.
- **Damn Vulnerable DeFi [67]**: Selected because of the great importance of understanding and addressing vulnerabilities in DeFi applications, which involve high-value transactions and digital assets. Vulnerabilities in these applications could allow attackers to steal funds, manipulate prices, perform flash lending attacks, among other attack vectors.
- **Capture the Ether [68]**: The main objective of Capture the Ether is to strengthen security and resilience in SC development by promoting better coding practices and awareness of existing vulnerabilities.

Analysis of Common Vulnerabilities in CTFs

Figure 4.2 shows a representation of the results obtained from the analysis of the challenges in Ethernaut. This analysis reveals that the most recurrent vulnerabilities identified in up to 3 different challenges are the following: unencrypted private

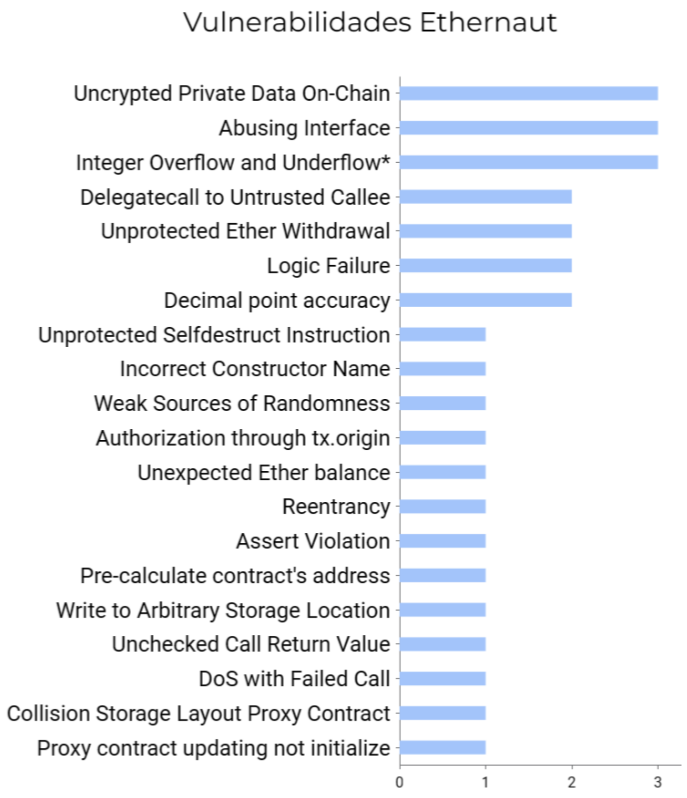


Figure 4.2: Vulnerability rate in the 27 Ethernaut challenges [9]

data in the chain, abuse of interfaces through inheritance between contracts and arithmetic integer overflow.

The results obtained after analysing the challenges present in Damn Vulnerable DeFi are shown in Figure 4.3. A significant correlation is revealed between finances and the most frequent vulnerabilities identified. It is relevant to highlight the recurrent use of Flash Loans as a means of exploiting vulnerabilities, present in up to 5 of the challenges analysed, as well as their use for manipulating oracles, which is observed in 4 of the challenges examined.

Figure 4.4 shows the results of the study of the challenges present in Capture The Ether. Revealing that the vulnerability that recurs with excellence in up to 6 different challenges is the pre-calculation of pseudo-random numbers due to weak randomness from string attributes.

By participating in these exercises, I gained practical experience with common vulnerabilities and learned how they are typically addressed in a controlled environment. This experience was instrumental in understanding the landscape of Ethereum security and identifying gaps that Faillapop could address.

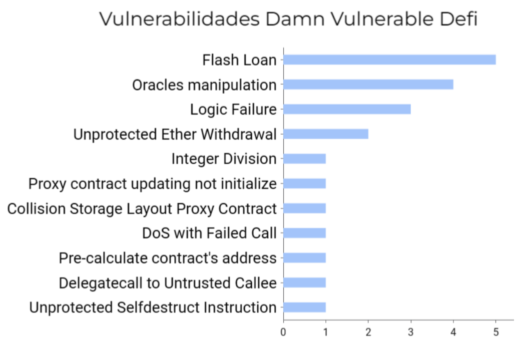


Figure 4.3: Vulnerability rate in Damn Vulnerable DeFi [9]

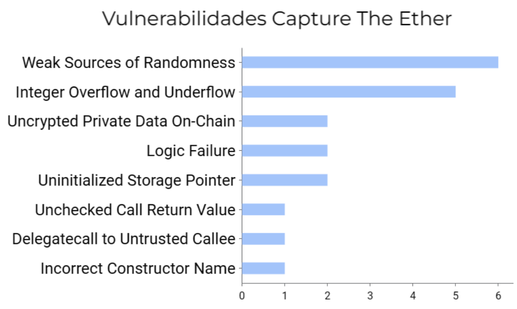


Figure 4.4: Vulnerability rate in Capture the Ether [9]

The result of this extensive engagement in CTFs and the analysis of the vulnerabilities encountered is published in [9].

4.2.3 Examination of Common Vulnerabilities in Public Contests and Exploited Protocols

In addition to participate in CTF exercises, analysing the most frequent vulnerabilities found in public contests and exploited protocols was essential to identify currently recurring patterns and issues that could serve as a basis for Faillapop's security strategy.

A study [7] of 516 exploitable bugs from 167 real-world contracts reported/-exploited in 2021-2022 revealed that over 80% of exploitable bugs are beyond existing detection tools, referred to as Machine Unauditable Bugs (MUBs). These MUBs were categorized into seven types, including price oracle manipulation, erroneous accounting, ID uniqueness violations, inconsistent state updates, privilege escalation, atomicity violations, and implementation-specific bugs.

This analysis underscored the importance of designing a protocol that not only educates but also challenges users to think critically about security. It also gave us an idea of what are the most relevant attack vectors in the ecosystem. This gave us a good perspective on how to develop Faillapop in a way that makes it a relevant educational tool in the current context.

Identification of Deficiencies in CTFs

Current CTF exercises, while addressing interesting and current vulnerabilities, often fall short in training students to detect them in complex real-world structures.

These exercises typically present vulnerabilities in simplified contexts, leading to a false sense of security among participants who might underestimate the complexity of real-world protocols.

Reflection on Faillapop Requirements

By synthesizing the findings from the Faillapop audit, CTF participation, and the study of vulnerabilities in real-world protocols, I developed a comprehensive perspective on Ethereum SC security. This holistic view allowed me to pinpoint critical areas for Faillapop's development and the specific vulnerabilities and bad practices to incorporate for educational purposes. The insights gained from this initial security analysis were crucial in shaping the project's direction, ensuring that Faillapop would serve as an effective tool for teaching and learning about SC security.

4.3 Protocol Development and Enhancements

Following the security analysis and context assessment, we established a clear roadmap for Faillapop's development. The primary focus of this phase was twofold: refining the existing protocol design and implementing new functionalities. By fine-tuning the current design, we aimed to establish a robust and secure base that would support the addition of new features. This involved thorough testing of the existing SCs to ensure their functionality. Only after confirming that the initial implementation was sound did we proceed to introduce new functionalities. Figure 4.5 shows the architecture of extended Faillapop [69], after my collaboration.

4.3.1 Introduction of New Functionalities

The addition of new functionalities was driven by the need to enhance the educational value of Faillapop and to simulate real-world scenarios more accurately. This required the development of new SCs and the integration of advanced features, all while maintaining a focus on security and best practices. Key implementations included:

New Contracts: CoolNFT and PowersellerNFT

To enhance Faillapop's realism and educational value, we developed two new SCs: CoolNFT and PowersellerNFT. Both contracts adhere to OpenZeppelin's

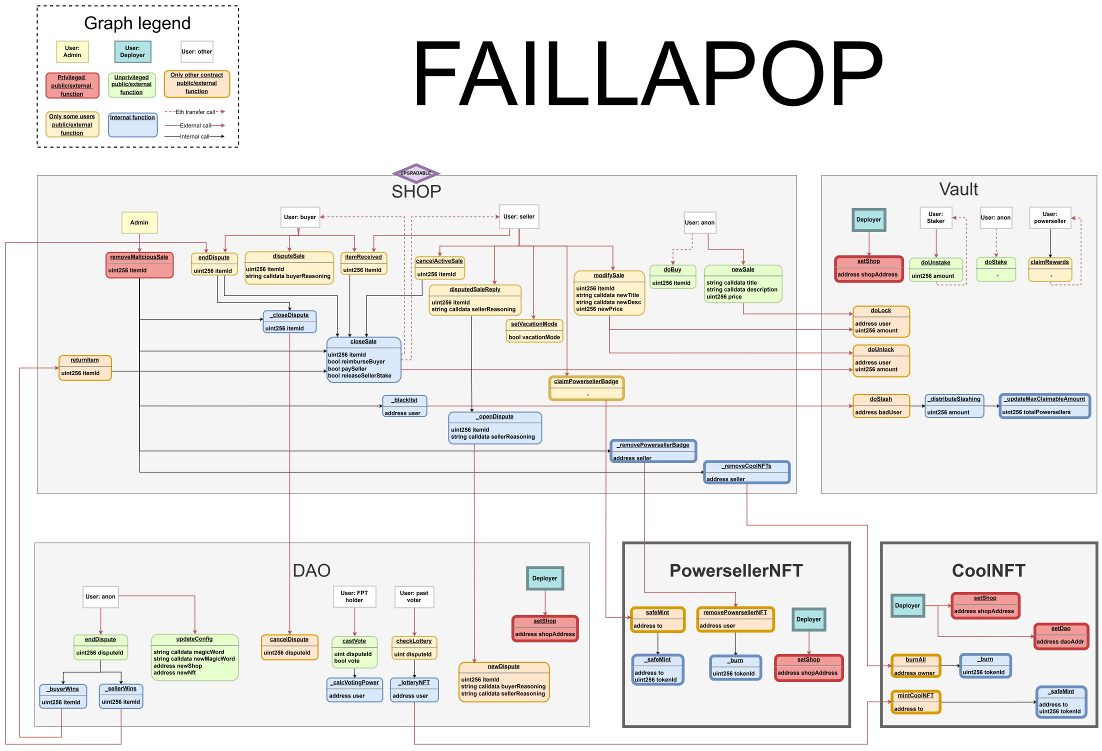


Figure 4.5: Faillapop's extended architecture

ERC721 standard [70], which defines a non-fungible token (NFT) interface for creating unique digital assets on the blockchain. The implementation of these contracts serves multiple purposes: rewarding user engagement, promoting platform integrity, and increasing the protocol's resemblance to real-world applications. Below is an explanation of each contract:

- **CoolNFT**: This contract incentivizes active participation within the decentralized autonomous organization (DAO) of the Faillapop protocol. Users who vote in DAO disputes and align with the winning side are eligible to participate in a lottery to mint a CoolNFT. Although the token lacks direct utility within the protocol, it acts as a badge of honor, fostering a sense of belonging and recognition. It may unlock benefits in future events within the Faillapop community.

- **PowersellerNFT**: This contract encourages consistent and honest participation among sellers. To qualify for a PowersellerNFT, a seller must have at least five weeks of active participation and a minimum of ten successful sales. Holding this token allows sellers to claim rewards from the vault, funded by penalizing malicious sellers. This mechanism promotes good behavior and discourages fraudulent activities, enhancing platform trust and reliability.

Integrating CoolNFT and PowersellerNFT into the Faillapop project significantly enhances its educational value. By incorporating these functionalities, the protocol mirrors real-world applications, providing students with a more realistic and comprehensive learning experience. Users not only learn about ERC721 standards and NFT creation but also gain insights into advanced concepts like non-transferability, reward systems, and automated revocation mechanisms.

Implementation of ERC1967 Proxy for Upgradable Shop Contract

To align Faillapop with real-world applications, we implemented an ERC1967 proxy to enable the upgradability of the Shop contract. This addition demonstrates a crucial aspect of modern SC development—ensuring that deployed contracts can, in a decentralized way, be updated to fix bugs, add features, or improve performance without disrupting the existing state and user interactions.

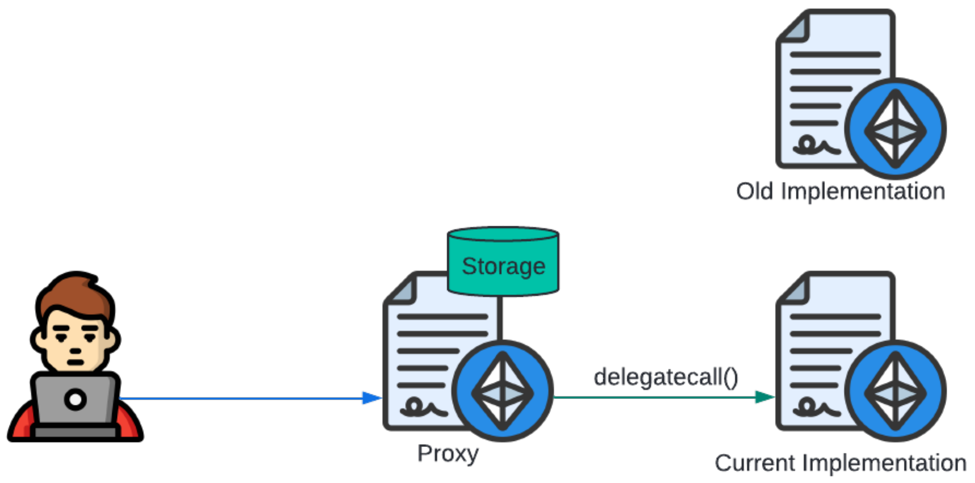


Figure 4.6: Simple Proxy Pattern

A proxy contract acts as an intermediary that forwards calls to a logic contract (the implementation contract), separating the contract's logic from its state, as shown in Figure 4.6. This separation allows for the logic to be upgraded independently of the state, which remains intact on the blockchain. Upgradable SCs are essential for long-term projects and applications handling significant value, as they offer the flexibility to adapt to new requirements and address security vulnerabilities without necessitating a complete redeployment.

OpenZeppelin [66] offers robust tools and libraries to facilitate the creation and management of upgradable contracts. Using OpenZeppelin's facilities, developers can leverage well-tested and secure implementations of proxy patterns, including the Transparent Proxy [71], UUPS (Universal Upgradeable Proxy Standard) [72], and Beacon Proxy [73]. We utilized OpenZeppelin's ERC1967 [74] standard which specifies the storage slots for the implementation address and an admin address, ensuring that these critical pieces of information are securely and predictably managed.

Delegatecall Mechanism

The `delegatecall` operation is critical for proxy functionality. It allows the logic contract's code to execute in the context of the proxy's storage, meaning state changes by the logic contract apply to the proxy's storage. This mechanism maintains the proxy's state while allowing the logic to be updated. However, it also introduces risks such as potential state misalignment, necessitating careful handling to ensure compatible storage layouts.

Implementation of the Proxy for the Shop Contract

To integrate the proxy pattern, the Shop contract, which contains the core business logic of the Faillapop marketplace, underwent several modifications. The primary change was replacing the constructor with an `initialize` function. This function serves the same purpose as a constructor but can be called only once, ensuring that the contract is correctly initialized when deployed through the proxy.

Proxy Setup and Initialization

The setup involved deploying the Shop contract as the logic contract and then deploying the proxy contract, pointing it to the Shop implementation. The `initialize` function of the Shop contract was called through the proxy to set up initial parameters and state variables. This setup allows users and other Faillapop contracts to interact with the Shop transparently, without needing to be aware of the proxy's presence.

Governance and Upgrade Process

The upgrade process is governed by the Faillapop DAO, ensuring that the community has control over significant changes to the protocol. Any user can propose an upgrade to the Shop contract, following this process:

1. **Proposal Submission**: Any user can submit a proposal for a Shop update through the DAO interface. The proposal must include the address of the new Shop implementation contract.
2. **Review Period**: The proposal enters a 1-day review period, allowing community members to examine the proposed changes and discuss their potential impact.
3. **Voting Period**: After the review period, the proposal enters a voting phase lasting a minimum of 3 days. During this time, DAO members can vote on whether to approve or reject the proposal.
4. **Proposal Cancellation**: Proposals can be cancelled at any time by the creator or the DAO admin, providing flexibility to withdraw or amend proposals based on community feedback.
5. **Execution Delay**: If the proposal is approved, there is a 1-day execution delay to provide an additional security buffer. After this period, any user can execute the proposal to upgrade the Shop contract.
6. **Upgrade Execution**: Upon execution, the proxy's implementation address is updated to the new Shop contract, enabling the new logic while preserving the existing state.

Security Considerations

Despite their advantages, proxy-based contracts are not without risks. Potential vulnerabilities include:

- **Unprotected Initializers**: Initializers must be invocable only once, similar to constructors, to prevent reinitialization attacks. This is typically ensured by using an *initializer* modifier that restricts the function to a single call. Failure to protect initializers can lead to severe security breaches.
- **Use of Constructors or non-constant initialised variables**: Any logic within the constructor of the logic contract is ineffective because the state

used will be that of the proxy, not the logic contract. Therefore, initial state variables should be set within an `initialize` function rather than the constructor.

- **Function Clashing**: Function clashing occurs when the function selectors in the proxy and the logic contract overlap, potentially leading to unintended behavior. Transparent Proxies, which route calls based on the `msg.sender`, can mitigate this issue by ensuring only administrative calls are handled by the proxy, while all other calls are delegated to the logic contract.
- **Storage Layout Collisions**: Storage layout collisions between the proxy and logic contracts can lead to critical vulnerabilities. For instance, if the storage layout of the logic contract changes in a way that conflicts with the proxy, it can result in incorrect behavior or exploitable security flaws. To avoid this, it is crucial to maintain a consistent storage layout across different versions of the logic contract. This can be achieved by following the append-only principle, where new state variables are only added to the end of the existing layout.

Integrating ERC1967 proxy and upgradable contracts into Faillapop enhances its educational value. Students gain hands-on experience with advanced SC patterns widely used in the industry. Understanding and implementing upgradable contracts prepare developers for real-world challenges, where flexibility and adaptability are crucial for long-term project success.

Commit-Reveal Scheme for Dispute Voting

The commit-reveal scheme is widely used in blockchain applications to ensure fair and secure voting, particularly in environments where transparency and public visibility are paramount, such as Ethereum. It is a two-phase process designed to ensure the confidentiality and integrity of votes.

1. **Commit Phase**: Voters submit a cryptographic commitment to their vote, which typically involves hashing the vote along with a secret nonce.
2. **Reveal Phase**: After the commit phase ends, voters reveal their original votes and the nonce used to generate the commitment. This allows anyone to verify that the revealed vote matches the commitment.

This mechanism ensures that users vote based on their genuine opinions rather than being influenced by the majority, which is particularly important when votes are tied to incentives such as participation in the CoolNFT lottery. Then, the new voting process includes:

1. **Dispute Creation**: When a dispute arises within the Faillapop ecosystem, a voting period is initiated. This period lasts for a minimum of 3 days, during which users can submit their votes.

2. **Commit Phase**: During the voting period, users submit their votes in the form of cryptographic commitments. Each commitment is a hash of the vote combined with a secret nonce, ensuring that the vote remains confidential until the reveal phase.

3. **Reveal Phase**: Following the commit phase, the reveal phase begins, lasting for at least 1 day. During this time, users reveal their votes and the corresponding nonces. The revealed information is then used to verify that the commitments match the votes.

4. **Closing the Dispute**: After the reveal phase, the dispute can be closed if the threshold of votes has been exceeded. If not, the voting process can extend for an additional 3 days to allow for further participation.

Incorporating the commit-reveal scheme into the Faillapop project provides users with practical experience in understanding cryptographic techniques used in real-world blockchain applications. This experience equip them with the knowledge and skills needed to design and deploy secure voting mechanisms in decentralized environments.

Comprehensive Documentation

In the development of the Faillapop project, comprehensive documentation has been produced to mirror the practices observed in real-world protocols. Documentation plays a critical role in the lifecycle of any software project. It serves as a formal record of the design, implementation, and intended functionality of the system. However, it is not uncommon to encounter slight deviations between the documentation of real-world projects and its actual codebase. Having said that, it is important that students learn the potential for discrepancies between documentation and the actual codebase.

To train participants effectively, the documentation of the Faillapop project not only provides essential insights into the system's design and functionality but also includes slight deviations from the actual codebase. This exercise teaches participants to critically evaluate the documentation, compare it with the code, and identify inconsistencies. This skill is crucial for security researchers who must validate that a project's implementation adheres to its documented intentions.

4.3.2 Enhancements in Code Quality

Enhancing code quality is paramount in developing reliable, secure, and maintainable SCs. For the Faillapop project, this involves implementing a series of improvements that not only align the project with industry standards but also enhance its educational value by exposing participants to best practices in blockchain development. This section delves into specific measures taken to improve the code quality of the Faillapop project.

Deployment Script Implementation

Initially, the Faillapop contracts faced an issue of circular dependencies in their constructors. For example, the Shop contract depended on the DAO contract's address in its constructor and vice versa, creating a deployment deadlock. This issue is commonly resolved by implementing an initialization function that is called post-deployment to provide the necessary addresses. To facilitate the deployment process and ensure the protocol can be deployed locally or on testnet for testing and exploitation attempts, a comprehensive deployment script has been developed. This script automates the deployment process, ensuring all contracts are deployed correctly and the protocol is ready for use. This approach mirrors real-world practices, enhancing the educational value of the project by demonstrating how deployment complexities are managed in professional environments.

Optimization Using Inline Assembly

Inline assembly allows developers to write low-level code directly within SCs. This technique is often used to optimize performance and reduce gas costs, which is crucial in the Ethereum ecosystem where transaction costs can be high. However, it introduces additional complexity and potential vulnerabilities.

In the Faillapop project, we have integrated two modifiers written in inline assembly. The first modifier checks if an address is zero, ensuring that invalid addresses are caught early in the execution process. The second, more complex modifier, is used within the Vault contract to handle specific operations more efficiently. Understanding and analyzing inline assembly is a valuable skill for security researchers, as it frequently appears in optimized contracts and requires careful scrutiny to ensure safety and correctness.

Implementation of Contract Interfaces

Interfaces play a critical role in SC development by defining the structure of the contracts and enabling other contracts or DApps to interact with them correctly. They provide a clear contract API, ensuring that external entities know exactly how to interact with the protocol's contracts. This is particularly important in decentralized ecosystems where interoperability and integration with other projects are common.

In the Faillapop project, all contracts have corresponding interfaces, clearly outlining the available functions and their expected inputs and outputs. This not only improves the modularity and readability of the code but also facilitates external integration and enhances the overall robustness of the protocol. By adhering to this practice, Faillapop aligns with real-world standards, making it an excellent educational tool for understanding the importance of interfaces in blockchain development.

Comprehensive Unit and Integration Testing

Testing is a cornerstone of robust software development, and its importance is magnified in the context of blockchain and SCs where immutability and the handling of valuable assets are involved. Unit tests focus on individual functions or components, ensuring they behave as expected in isolation. Integration tests, on the other hand, verify that different components of the system work together correctly.

In Faillapop, we have implemented extensive unit and integration tests to validate the correctness and reliability of the contracts. Each function is tested for both expected (successful) cases and edge (failure) cases, ensuring comprehensive coverage. This methodology ensures that not only does each component function correctly, but the interactions between components are also verified.

From an auditing perspective, tests serve multiple purposes:

- **Understanding Interactions**: They help auditors understand how different contracts interact with each other, providing insight into the system's behavior.
- **Validating Functionality**: Tests can be used to validate that the implementation matches the documented intentions.
- **Identifying Vulnerabilities**: Writing and running tests can help identify potential vulnerabilities or logic errors that might not be apparent from the code

alone.

However, due to time constraints, many auditors may prioritize direct code review over extensive test analysis. In Faillapop, we emphasize the importance of testing by including comprehensive test coverage and even introducing subtle pitfalls within the test suite. This encourages participants to thoroughly analyze the tests, reinforcing the lesson that thorough testing is crucial for securing a protocol.

Adherence to Solidity Best Practices

Adhering to best practices in Solidity development is essential for creating secure, maintainable, and scalable SCs. These practices are widely followed by leading blockchain companies and projects.

Throughout the Faillapop project, we have diligently followed these best practices, some of which are:

- **Correct Importing of Dependencies**: Ensuring that all dependencies are imported correctly and efficiently.
- **Contract Layout Order**: Following a consistent and logical order in contract layout, which typically includes state variables, events, modifiers, functions, and fallback functions.
- **CamelCase Naming Convention**: Using camelCase for variable and function names to maintain readability and consistency.
- **Visibility Declarations**: Explicitly declaring the visibility of variables and functions (public, internal, external, private) to enhance security and clarity.
- **Constants in Capital Letters**: Defining constants in all capital letters to differentiate them from regular variables and to signify their immutable nature.

These practices contribute to the overall quality and professionalism of the codebase, making it easier for developers and auditors to understand and work with the contracts. Additionally, by implementing these improvements, the Faillapop project not only mirrors real-world protocols but also serves as a comprehensive educational tool. Participants learn to appreciate the importance of code quality, thorough testing, and adherence to best practices, all of which are essential skills for successful blockchain development and security auditing.

4.4 Deliberate Introduction of Vulnerabilities

Faillapop is a vulnerable-by-design protocol, specifically crafted as an educational tool to help students and professionals understand and identify common security issues in SCs. To achieve this, intentional vulnerabilities have been introduced into the protocol.

4.4.1 Rationale for Introducing Vulnerabilities

The vulnerabilities introduced in Faillapop are carefully chosen to serves several educational purposes:

- **Educational Realism**: Real-world protocols often contain flaws due to oversight, complexity, or evolving threats. By intentionally incorporating current vulnerabilities, Faillapop provides a realistic environment in which participants can practice identifying and mitigating problems that affect today's world.
- **Comprehensive Learning**: By dealing with a range of vulnerabilities, participants develop a thorough understanding of various security issues and how they can manifest in interconnected SCs.
- **Enhanced Skill Development**: Participants learn to navigate the intricate web of dependencies and potential vulnerabilities, which is a crucial skill for real-world blockchain development and security auditing.
- **Practical Experience**: The exercise provides hands-on experience in detecting and addressing security risks, bridging the gap between theoretical knowledge and practical application.
- **Engagement and Motivation**: The challenge of finding and fixing vulnerabilities adds an element of gamification, making the learning process more engaging and motivating.

4.4.2 Types of Vulnerabilities Introduced

The vulnerabilities introduced in the Faillapop protocol encompass a variety of common issues found in SCs. These include:

- **Reentrancy Attacks**: Allow an attacker to repeatedly call a function before the previous execution completes, potentially leading to fund depletion.

- **Unchecked Call Return Values**: Assuming external calls succeed without checking their return values can lead to unexpected behavior and security risks.
- **Integer Overflow and Underflow**: Arithmetic operations that exceed the maximum or minimum values can result in incorrect calculations and exploitable conditions.
- **Access Control Issues**: Improper access control mechanisms can allow unauthorized users to perform restricted actions, compromising the protocol's integrity.
- **Denial of Service (DoS) Vulnerabilities**: These can prevent the contract from functioning correctly by consuming excessive gas or blocking critical operations through malicious input.

By intentionally incorporating these vulnerabilities, Faillapop offers a robust educational framework that mirrors the complexity and interdependencies found in real-world DApps. This approach not only enhances participants' technical skills but also fosters a deeper appreciation for the intricacies and importance of SC security.

Part IV

Conclusion

Chapter 5

Final Evaluation and Future Directions

In conclusion, this work discussed different aspects in the field of blockchain, more exactly, Ethereum. It specifically explains basic principles, security issues, and the conception of educational materials to confront the latter. Ethereum's decentralized, immutable, transparent, and deterministic nature presents unique security concerns. Critical vulnerabilities, such as reentrancy, transaction ordering dependencies, arithmetic flaws, and mishandled exceptions, pose significant threats to the integrity and reliability of SCs. Representing a significant challenge to broader adoption and trust in these systems.

The complex nature of such security issues highlight the need for robust educational resources. Traditional educational methods often fall short in providing the depth and realism necessary for effective learning. This gap underscores the necessity for innovative tools that offer practical, hands-on experience with realistic blockchain scenarios and common vulnerabilities.

Faillapop aims to address these needs providing participants with a decentralized application environment composed of interconnected SCs, challenging them with scenarios that reflect real-world complexities. By intentionally introducing vulnerabilities such as reentrancy attacks, unchecked call return values, and integer overflow/underflow issues, Faillapop fosters critical thinking and problem-solving skills. This practical approach equips users with the knowledge and experience necessary to excel in blockchain security auditing and development. This project also serves as a valuable resource for academic institutions and companies aiming to upskill their teams.

Going forward, we commit to to keeping Faillapop updated with the latest vulnerabilities and best practices. As the blockchain ecosystem evolves, so must our educational resources. We aim to continuously enhance Faillapop by integrating new functionalities and refining its educational content to ensure it remains a cutting-edge tool for Ethereum security education. Our goal is to promote Faillapop widely among developers, companies, and educational institutions, thereby contributing to the overall enhancement of Ethereum security through high-quality, hands-on training.

Additionally, we plan to conduct cybersecurity workshops at relevant industry events, further extending the reach and impact of Faillapop. These workshops will provide an opportunity for direct engagement with the community, fostering a deeper understanding of blockchain security challenges and solutions. Through these efforts, we aspire to foster a more secure and resilient blockchain ecosystem, empowering developers and auditors to address the dynamic challenges of DApps effectively.

Bibliography

[1] K. Fanning and D. P. Centers. Blockchain and its coming impact on financial services. *Journal of Corporate Accounting & Finance*, 27(5):53–57, 2016.

[2] M. Attaran. Blockchain technology in healthcare: Challenges and opportunities. *International Journal of Healthcare Management*, 15:70–83, 11 2020.

[3] H.-N. Dai, Z. Zheng, and Y. Zhang. Blockchain for internet of things: A survey. *IEEE Internet of Things Journal*, 6(5):8076–8094, 2019.

[4] S. A. Abeyratne and R. P. Monfared. Blockchain ready manufacturing supply chain using distributed ledger. *International Journal of Research in Engineering and Technology*, 5(9):1–10, 2016.

[5] Ethereum charts and statistics — etherscan. `https://etherscan.io/chart/tx`. Accessed: 2024-05-30.

[6] IMMUNEBYTES. Euler finance hack—mar 13, 2023-detailed hack analysis. `https://www.immunebytes.com/blog/euler-finance-hack-mar-13-2023-detailed-hack-analysis/`, 2023.

[7] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. Demystifying exploitable bugs in smart contracts. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 615–627, 2023.

[8] Tanusree Sharma, Zhixuan Zhou, Andrew Miller, and Yang Wang. A mixed-methods study of security practices of smart contract developers. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, SEC '23, USA, 2023. USENIX Association.

[9] JM. Tapia, M. López, I. Agudo, and JC. Ramírez. *Análisis de las vulnerabilidades en Smart Contracts: desafíos CTF para mejorar la seguridad*, pages 118–121. Jornadas de Ingeniería Telemática (JITEL), 2023.

[10] A. Averin and O. Averina. Review of blockchain technology vulnerabilities and blockchain-system attacks. In *Proc. Int. Multi-Conf. Ind. Eng. Mod. Technol.*, pages 1–6, 2019.

[11] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[12] C. K. Frantz and M. Nowostawski. From institutions to code: Towards automated generation of smart contracts. In *Proc. IEEE 1st Int. Workshops Found. Appl. Self Syst.*, pages 210–215, 2016.

[13] A. Mense and M. Flatscher. Security vulnerabilities in ethereum smart contracts. In *Proc. 20th Int. Conf. Inf. Integr. Appl. Services*, pages 375–380, 2018.

[14] D.L. Chaum. Computer Systems Established, Maintained and Trusted by Mutually Suspicious Groups. Technical report, University of California, Electronics Research Laboratory, 1979.

[15] D. Bayer, S. Haber, and W.S. Stornetta. Improving the efficiency and reliability of digital timestamping. *Sequences II*, pages 329–334, 1993.

[16] N. Szabo. Formalizing and securing relationships on public networks. *First Monday*, 1997.

[17] B. Carlsson and R. Gustavsson. The rise and fall of napster - an evolutionary approach. In *Active Media Technology. AMT 2001*. Springer, 2001.

[18] D. Bayer, S. Haber, and W.S. Stornetta. Improving the efficiency and reliability of digital timestamping. *Sequences II*, pages 329–334, 1993.

[19] H. Finney. Rpow-reusable proofs of work. `https://nakamotoinstitute.org/finney/rpow/index.html`, 2004.

[20] R. Pass, L. Seeman, and A. Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Annual International conference on the Theory and Applications of Cryptographic Techniques*, pages 643–6, 2017.

[21] Shikah J. Alsunaidi and Fahd A. Alhaidari. A survey of consensus algorithms for blockchain technology. In *2019 International Conference on Computer and Information Sciences (ICCIS)*, pages 1–6, 2019.

[22] Ethereum charts and statistics — etherscan. `https://etherscan.io/stat/supply`. Accessed: 2024-05-30.

[23] William E. Bodell III, Sajad Meisami, and Yue Duan. Proxy hunting: Understanding and characterizing proxy-based upgradeable smart contracts in blockchains. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1829–1846, 2023.

[24] Taotao Wang, Chonghe Zhao, Qing Yang, Shengli Zhang, and Soung Chang Liew. Ethna: Analyzing the underlying peer-to-peer network of ethereum blockchain. *IEEE Transactions on Network Science and Engineering*, 8(3):2131–2146, 2021.

[25] Darko Čapko, Srđan Vukmirović, and Nemanja Nedić. State of the art of zero-knowledge proofs in blockchain. In *2022 30th Telecommunications Forum (TELFOR)*, pages 1–4, 2022.

[26] Samuel Steffen, Benjamin Bichsel, Roger Baumgartner, and Martin Vechev. Zeestar: Private smart contracts by homomorphic encryption and zero-knowledge proofs. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 179–197, 2022.

[27] G.M. Bertoni, Joan Daemen, and Michael Peeters. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3, 01 2009.

[28] G.M. Bertoni, Joan Daemen, and Michael Peeters. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3, 01 2009.

[29] V. Buterin. A Next-Generation Smart Contract and Decentralized Application Platform, 2013.

[30] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Proj. Yellow Pap.*, 151:1–32, 2014.

[31] H. Watanabe, S. Fujimura, A. Nakadaira, Y. Miyazaki, A. Akutsu, and J. J. Kishigami. Blockchain contract: A complete consensus using blockchain. In *Proc. IEEE 4th Global Conf. Consum. Electron. (GCCE)*, pages 577–578, 2015.

[32] A. Pinna, S. Ibba, G. Baralla, R. Tonelli, and M. Marchesi. A massive analysis of ethereum smart contracts empirical study and code metrics. *IEEE Access*, 7:78194–78213, 2019.

[33] Chainlink oracles. `https://drive.google.com/file/d/1RmVHcOAD8taIa7kTYGef-Y-4-wejVRJm/view`.

[34] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Security Symposium (USENIX Security 01)*, 2001.

[35] F. J. Serna. The info leak era on software exploitation. Black Hat USA, 2012.

[36] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *international conference on Information security*. Springer, 2010.

[37] Yongfeng Huang, Yiyang Bian, Renpu Li, J Leon Zhao, and Peizhong Shi. Smart contract security: A software lifecycle perspective. *IEEE Access*, 7:150184–150202, 2019.

[38] F. Ma, Y. Fu, M. Ren, M. Wang, Y. Jiang, K. Zhang, H. Li, and X. Shi. Evm: From offline detection to online reinforcement for ethereum virtual machine. In *Proc. IEEE 26th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, pages 554–558. IEEE, 2019.

[39] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, and X. Luo. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In *Information Security Practice and Experience (Lecture Notes in Computer Science)*, volume 10701, pages 3–24. Springer, 2017.

[40] Chainlink oracles. `https://chain.link/education-hub/maximal-extractable-value-mev`.

[41] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe. Reguard: Finding reentrancy bugs in smart contracts. In *Proc. 40th Int. Conf. Softw. Eng., Companion*, pages 65–68. ACM, 2018.

[42] M. Wohrer and U. Zdun. Smart contracts: security patterns in the ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 2–8, 2018.

[43] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee. Systematic review of security vulnerabilities in ethereum blockchain smart contract. *IEEE Access*, 10:6605–6621, 2022.

[44] J. Krupp and C. Rossow. teEther: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1317–1333, 2018.

[45] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss. Security analysis methods on ethereum smart contract vulnerabilities: A survey. *arXiv*, 2019.

[46] X. Liu, R. Chen, Y.-W. Chen, and S.-M. Yuan. Off-chain data fetching architecture for ethereum smart contract. In *2018 International Conference on Cloud Computing, Big Data and Blockchain (ICCBB)*, pages 1–4, 2018.

[47] Chainlink verifiable random function. `https://blog.chain.link/chainlink-vrf-on-chain-verifiable-randomness/`.

[48] Vrfconsumerbasev2 contract. `https://github.com/smartcontractkit/chainlink/blob/develop/contracts/src/v0.8/vrf/VRFConsumerBaseV2.sol`.

[49] Chainlink vrf. Available online: https://docs.chain.link/vrf.

[50] Randao. Available online: https://github.com/randao/randao (accessed on 19 January 2022), 2019.

[51] A. Mense and M. Flatscher. Security vulnerabilities in ethereum smart contracts. In *Proceedings of the 20th International Conference on Information Integration and Web-Based Applications and Services*, pages 375–380. ACM, 2018.

[52] R. Yang, T. Murray, P. Rimba, and U. Parampalli. Empirically analyzing ethereum's gas mechanism. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 310–319. IEEE, 2019.

[53] Home — uniswap protocol. Available online: https://uniswap.org/.

[54] Flash loans attacks. `https://chain.link/education-hub/flash-loans`.

[55] Daniel Perez and Benjamin Livshits. Smart contract vulnerabilities: Does anyone care. *arXiv preprint*, pages 1–15, 2019.

[56] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2016.

[57] J. Feist, G. Grieco, and A. Groce. Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019.

[58] B. Mueller. Smashing ethereum smart contracts for fun and real profit. Available online: https://github.com/muellerberndt/smashing-smart-contracts, 2018.

[59] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, pages 1186–1189, 11 2019.

[60] trailofbits. Echidna, a smart fuzzer for ethereum. Available online: https://blog.trailofbits.com/2018/03/09/echidna-a-smart-fuzzer-for-ethereum/ Accessed: 2023-05-31.

[61] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, pages 530–541. IEEE, 2020.

[62] Asem Ghaleb and Karthik Pattabiraman. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 415–427. ACM, 2020.

[63] Asem Ghaleb. Towards effective static analysis approaches for security vulnerabilities in smart contracts. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–5. IEEE, 2022.

[64] Faillapop's initial stage of development. `https://github.com/jcsec-security/solidity-security-course-resources/tree/157076be82e59bc5a8b2ab48c9e8d4ccfb3221dd/faillapop`.

[65] Ethernaut. `https://ethernaut.openzeppelin.com`.

[66] Openzeppelin. `https://openzeppelin.com`.

[67] Damn vulnerable defi. `https://www.damnvulnerabledefi.xyz`.

[68] Capture the ether. `https://capturetheether.com`.

[69] Extended faillapop. `https://github.com/jcsec-security/solidity-security-course-resources/tree/main/faillapop`.

[70] Erc721. `https://docs.openzeppelin.com/contracts/5.x/erc721`.

[71] Transparent proxy pattern. `https://blog.openzeppelin.com/the-transparent-proxy-pattern`.

[72] Uups proxy. https://docs.openzeppelin.com/contracts/5.x/api/proxy#UUPSUpgradeable.

[73] Beacon proxy. https://docs.openzeppelin.com/contracts/5.x/api/proxy#beacon.

[74] Erc1967. https://docs.openzeppelin.com/contracts/5.x/api/proxy#erc1967.