Research & Development

PER2023-051

Self-modifiable smart contracts applied to FinTech/Defi

Master 2 EIT DIGITAL (FinTech), DS4H

Students: Mario Sorrentino, Giovanni Rigotti, Simone Bianchin

Supervisor: Luigi Liquori

Nice, 29/02/2024

Contents

1 Introduction

1.1 Blockchain Overview

Blockchain technology represents a revolutionary approach to how information is stored, verified, and exchanged on a digital platform. It operates as a decentralized and distributed digital ledger, organizing data into blocks, which are then chronologically linked together in a chain, thus the name *"blockchain"*. Each block contains a series of transactions along with a cryptographic reference, known as a hash, to the preceding block in the chain. This unique cryptographic structure ensures a high level of security; modifying even a minor detail in a single block necessitates recalculating all subsequent block hashes, making the blockchain incredibly secure and resistant to fraud and manipulation.

The birth of blockchain technology is closely tied to the creation of *Bitcoin* by the enigmatic figure *Satoshi Nakamoto in 2008*, a period marked by growing distrust in traditional financial institutions following the financial crisis. Blockchain was conceived as a solution to the lack of transparency and centralized control characteristic of the banking system, proposing a more democratic, transparent, and secure financial system. Unlike traditional financial systems where transactions and data are controlled by a central authority, blockchain operates through a distributed network of participants, referred to as nodes. These nodes collectively verify and validate transactions, thereby eliminating central points of failure and enhancing the network's resilience against attacks, censorship, or manipulation.

One of the blockchain's key features is its decentralized nature, which not only ensures that all participants have access to consistent and transparent information but also significantly increases the system's overall resilience. Even if some nodes were to be compromised, the blockchain system would remain stable and functional. This distributed verification process ensures that no single entity has control over the entire network, thereby increasing security and trust among participants.

Blockchain's impact extends far beyond the realm of cryptocurrencies; its potential applications span various sectors, promising to revolutionize not just the financial industry but also other areas by providing a means to securely manage and protect data without relying on central authorities. Currently valued at *$32.69 billion*, the global blockchain market is expected to reach *$162.84 billion by 2027*, with a compound annual growth rate *(CAGR) of 70.8%*. [2] These data underscore the growing focus on blockchain as a backbone for innovative solutions in the coming years.

1.2 Ethereum

Ethereum is a public, permissionless blockchain, second in market capitalization only to Bitcoin. It utilizes a peer-to-peer network to securely execute and verify application code, known as **smart contracts (SCs)**. This platform is at the forefront of enabling the creation of decentralized applications (dApps) and organizations, free from censorship and centralized control. At its core, Ethereum is powered by the **Ethereum Virtual Machine (EVM)**, a universal computation engine agreed upon by all participants in the Ethereum network. The EVM represents a pivotal innovation, allowing every network participant to maintain a copy of its state and to request computational operations through transactions. The execution of these transactions by the EVM results in state changes that are recorded on the blockchain, ensuring consistency and agreement across all nodes. This ledger not only captures all transactional history but also the current

state of the EVM, safeguarded by robust cryptographic mechanisms. These mechanisms validate transactions and secure them against tampering, while also enforcing execution permissions to protect digital assets from unauthorized access.

Ethereum's adoption of a proof-of-stake consensus mechanism marks a significant departure from the traditional proof-of-work model. In this system, individuals can participate in the network's security and consensus by staking ETH, Ethereum's native cryptocurrency, and running validator software. Validators are selected randomly to propose new blocks, which are then verified by other validators and added to the blockchain. This process is incentivized through a system of rewards and penalties, encouraging validators to act honestly and maintain network integrity.

1.3 Smart Contracts

Smart contracts represent a significant leap forward in the utilization of blockchain technology, fundamentally altering how agreements are executed across various domains. Introduced by Nick Szabo in 1996, these computer programs automate negotiation and execution processes between parties, improving the transparency, efficiency and security of business transactions and eliminating the need for trusted intermediaries.

Embedded within blockchain code through simple '*if... then...*' statements, smart contracts autonomously execute when predetermined conditions are met. This process is facilitated by a decentralized network of computers, or nodes, that collectively execute and validate smart contracts. Each state-changing transaction resulting from smart contract execution is recorded on the blockchain to ensure traceability and immutability.

The growing adoption of smart contracts is set to revolutionize industries such as finance, real estate, healthcare and supply chain management, driven by a growing demand for digitization, automation and secure transactions. **However, the journey toward widespread acceptance is marked by several challenges, primarily due to the immutable nature of smart contracts.**

1.4 Solidity

Solidity is a high-level, statically typed programming language designed for developing smart contracts on the Ethereum blockchain. With a syntax resembling C++, Python, and JavaScript, it prioritizes accessibility and ease of use. By integrating object-oriented programming (OOP) principles such as inheritance, encapsulation, and polymorphism, Solidity enables the development of complex, yet efficient, smart contracts that can underpin a wide array of decentralized applications (DApps) and complex financial mechanisms.

In addition to Solidity, the smart contract landscape offers several alternatives. *Vyper*, for example, stands as a viable choice within the Ethereum ecosystem, emphasizing security and simplicity of code. Shifting the focus beyond Ethereum, we find languages such as *Clarity* for Stacks, *Rust* for Solana and *Michelson* for Tezos. Each has unique features designed to optimize efficiency, security, and ease of development based on the specifics of the respective technology. This diversity of languages reflects the continuing evolution of the smart contract industry, underscoring the importance of informed choice based on design and security requirements.

1.5 Error Handling

Solidity offers four main error handling techniques: '*revert*', '*require*', '*assert*', and '*try/catch*', each designed to manage errors by reverting any state changes within the current call. Solidity distinguishes between two types of errors: '*Error(string)*' for common issues and '*Panic(uint256)*' for critical bugs that indicate flawed code.

- **assert** is intended for checking internal errors and invariants. Its use implies that the condition should never fail unless there's a bug in the contract code, making it crucial for detecting internal inconsistencies.
- **require** is primarily used for validating input, contract states, or outputs from external calls that are only verifiable at runtime. Failing a 'require' check triggers an 'Error(string)' error, effectively reverting the transaction unless conditions are satisfactorily met.
- **revert** explicitly cancels the current call and throws an exception to signal error conditions, optionally providing a reason via 'Error(string)' error signature, thus allowing for detailed error messages.
- **try/catch** enables handling exceptions from external contract calls or during contract creation. It captures errors allowing the contract to react to failures in calls to external functions or contracts. The 'try' block precedes the external call, while the 'catch' clause handles errors, distinguishing between 'Error(string)' and low-level exceptions with 'catch (bytes memory lowLevelData)' or a generic 'catch' for any error type.

```
function processWithTryCatch() public {
        try this.doSomething() {
            // Execution if doSomething() doesn't throw an error
        } catch Error(string memory errorMessage) {
            // Handling error if doSomething() throws a known
               error
            emit ErrorOccurred(errorMessage);
        } catch (bytes memory errorData) {
            // Handling other types of errors
            // This catch block is executed if an unknown error
               occurs
            // and errorData contains the low-level error
               information
        }
    }
```

2 Context

2.1 Problem Definition

The use of Turing-complete languages in crafting smart contracts, while offering vast possibilities, also opens the door to vulnerabilities. Indeed, attacks targeting smart contracts are on the rise, causing an estimated $6.45 billion in financial losses. [8] Attacks such as the **Decentralized Autonomous Organization (DAO)** and the Parity Wallet hack have cost millions of dollars simply as a consequence of naïve bugs in the smart contract code. The DAO incident, in particular, underscored the inherent risks, resulting in the theft of millions of dollars and prompting a controversial decision to fork the Ethereum blockchain in an effort to recover the lost funds. This move, while partially successful in restoring the assets, sparked a significant debate within the community, ultimately culminating in a schism that gave birth to two distinct blockchains: *Ethereum* and *Ethereum Classic*.

The results of a scan of almost one million contracts on the Ethereum blockchain found **34,200** subject to vulnerabilities. These contracts are vulnerable to attacks that could cause ETH (Ethereum cryptocurrency) theft, freezing or deletion of digital assets, and consequently financial damage and loss of control for users. [13]

The large number of vulnerable smart contracts and the serious consequences that can result highlight the urgent need for concrete solutions.

2.2 Challenges

Immutability, a key feature of blockchain technology, ensures that once a smart contract is deployed on the blockchain, it cannot be altered or revised. While this feature provides security and trust by preventing tampering after deployment, it also introduces a rigid inflexibility that poses challenges in dynamic real-world scenarios.

Renegotiation of agreements is a fundamental pillar of contract law. In traditional legal systems, contracting parties can modify the terms of a contract in case of circumstantial changes or to correct errors. This flexibility allows agreements to be adjusted to new information or changed conditions, ensuring their appropriateness and fairness over time.

However, smart contracts, by their immutable nature once deployed on the blockchain, do not benefit from the same flexibility. In fact, the only way to fix a bug or change contract terms would involve creating a new contract and having all participants agree to migrate to it, a process that can be cumbersome and not always feasible.

In addition to the inherent difficulties associated with the immutability of smart contracts, the process of updating them to include new features or to adapt to changing regulatory environments presents additional challenges. Whereas in traditional contracts, such changes are relatively straightforward, provided there is consensus among the parties involved, the rigidity of smart contracts means that developers must anticipate all possible future scenarios and incorporate mechanisms for updates or changes from the outset, a prohibitive task given the complexity and unpredictability of real-world applications.

Can innovative solutions, such as "Upgradable Smart Contracts", effectively address this issue? Fortunately, the development of these adaptable contracts offers an innovative approach to

mitigate these challenges, paving the way for greater flexibility in the deployment of smart contracts.

2.3 Upgradable Smart Contracts

What is a smart contract upgrade?
A smart contract upgrade is an action that can arbitrarily change the code executed in an address while preserving storage and balance. [19]

Unlike traditional smart contracts, whose logic is immutable once implemented in the blockchain, upgradable smart contracts offer the flexibility to modify and improve the code in response to discovered vulnerabilities or changes in requirements.
The possibility of updating smart contracts is especially valuable given the flaws in current security tools. Despite precautionary measures, some types of bugs go undetected when the contract is released. Upgradability allows developers to take prompt action if bugs are discovered. To empower this functionality, "modern" smart contracts use advanced software techniques, raising questions about how Upgradability is handled and who has permission to make changes.

2.3.1 Community Opinion

On the topic, the blockchain community shows contrasting opinions on this enhancement, reflecting a vibrant debate that balances fundamental principles and practical needs.

On one hand, some critics point out how the introduction of the capacity to upgrade may undermine the element of immutability that underlies the trust in blockchain technology. This feature, for many, is the pillar on which the security and inviolability of digital transactions and contracts is built, ensuring that once written on the ledger, the content cannot be altered.

On the other hand, supporters of this innovation emphasize the security and flexibility benefits that upgradable smart contracts can offer. They argue that the ability to fix bugs and vulnerabilities post-deployment represents a significant improvement in the resilience of these blockchain applications. The ability to adapt and respond to emerging threats by adding functionality or optimizing performance can, according to this perspective, strengthen the security of the blockchain ecosystem as a whole.

Within this context of divergent opinions, the challenge is to implement mechanisms for updating that keep the principles of decentralization and transparency intact, without compromising user trust.

3 Related Work

Smart contract updates are not a new concept for Ethereum developers. In fact, several implementations for smart contract updates have been proposed, either as a safeguard to implement a fix in case of vulnerabilities or as a means to iteratively develop a system by progressively adding new features.

There are many strategies for modifying a system without requiring a full upgrade.

3.1 ”AS-IS” Workflow

An easy solution is to change the system through a **migration**; here's what you'd need to do to fix a bug in a contract you cannot upgrade:

- Deploy a new version of the contract
- Manually migrate all state from the old one contract to the new one (which can be very expensive in terms of gas fees!)
- Update all contracts that interacted with the old contract to use the address of the new one
- Reach out to all your users and convince them to start using the new deployment (and handle both contracts being used simultaneously, as users are slow to migrate)

The ”as-is” workflow for modifying a smart contract on platforms like Ethereum presents a pragmatic approach to addressing the need for updates, whether for fixing vulnerabilities or adding new features. However, this process entails significant challenges and drawbacks that can impact the overall efficiency, cost, and user experience of the blockchain application. Essentially, it is not an optimal solution!

3.2 OpenZeppelin

OpenZeppelin, a leading developer of Smart Contract libraries, offers multiple solutions for the development of Upgradable Smart Contracts, all based on the concept of **proxy patterns**: The basic idea is using a proxy for upgrades. The first contract is a simple wrapper or ”proxy” which users interact with directly and is in charge of forwarding transactions to and from the second contract, which contains the logic. The key concept to understand is that the logic contract can be replaced while the proxy, or the access point is never changed. Both contracts are still immutable in the sense that their code cannot be changed, but the logic contract can simply be swapped by another contract. The wrapper can thus point to a different logic implementation and in doing so, the software is ”upgraded”. [15]

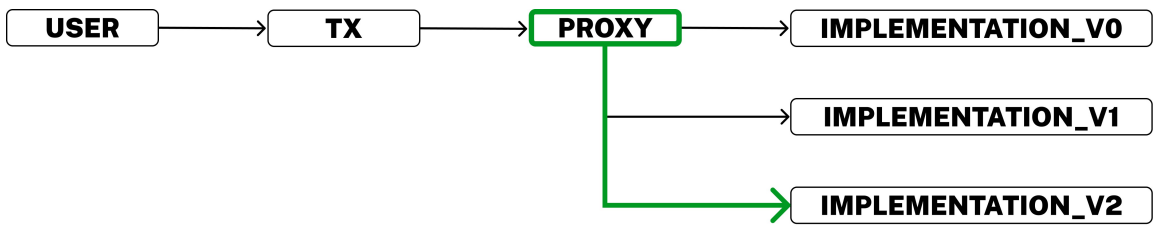


Figure 1: Proxy pattern

The most immediate problem that proxies need to solve is how the proxy exposes the entire interface of the logic contract without requiring a one to one mapping of the entire logic contract's interface. That would be difficult to maintain, prone to errors, and would make the interface itself not upgradable. Hence, a dynamic forwarding mechanism is required. A very important thing to note is that the code makes use of the EVM's delegatecall opcode which executes the callee's code in the context of the caller's state. That is, the logic contract controls the proxy's state and the logic contract's state is meaningless. Thus, the proxy doesn't only forward transactions to and from the logic contract, but also represents the pair's state. The state is in the proxy and the logic is in the particular implementation that the proxy points to.

```
assembly {
  // (1) copy incoming call data
  calldatacopy(0, 0, calldatasize())

  // (2) forward call to logic contract
  let result := delegatecall(gas(), implementation, 0,
     calldatasize(), 0, 0)

  // (3) retrieve return data
  returndatacopy(0, 0, returndatasize())

  // (4) forward return data back to caller
  switch result
  case 0 {
      revert(0, returndatasize())
  }
  default {
      return(0, returndatasize())
  }
}
```

Based on the mechanism presented, here are a series of standards proposed by the Ethereum community as solutions for implementing Upgradable Smart Contracts:

- **ERC-1167**, or the **Minimal Proxy Contract** standard, streamlines the creation of proxy contracts on Ethereum by using a uniform bytecode redirection technique. This ensures that contracts redirect consistently, making them easily identifiable and reliable for users and third-party services like Etherscan. It allows for the detailed inspection of redirecting contract's bytecode to verify the behavior and integrity of the destination code, based on verified sources or audits. This standard efficiently forwards all calls and gas to the target implementation contract, ensuring that any reverts and messages are accurately relayed back to the caller. [3]

- **ERC-1967** introduces the **Proxy Storage Slots** standard to efficiently and securely manage delegated proxy contracts, which are widely used in the Ethereum ecosystem to facilitate smart contract upgradability and optimize gas consumption. These proxy contracts delegate code execution to a logical contract (also known as an implementation contract) through the use of the '*delegatecall*' command. This approach allows proxies to maintain persistent state (storage and balance) while code execution is delegated to the logical contract. To prevent conflicts in storage usage between the proxy and the logical contract, the address of the logical contract is saved in a specific storage slot,

such as '0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc' in OpenZeppelin smart contracts, which is guaranteed never to be allocated by the compiler. ERC-1967 then proposes a set of standardized slots to store the proxy information. This allows clients such as block explorers to correctly extract and display this information to end users, and logical contracts to optionally interact with this information. [5]

- **ERC-2535**, also known as **Diamonds, Multi-Faced Proxy** introduces a standard for modular smart contract systems that provides a flexible and scalable framework for developing smart contracts on Ethereum. Unlike traditional smart contracts, Diamonds allow post-deployment upgrades and extensions, effectively overcoming the size limits imposed by the platform. In detail, a Diamond is characterized by the ability to aggregate external functions provided by specific contracts called "facets." The latter are independent contracts that can share internal functions, libraries and state variables, thus facilitating advanced modularity and effective code management. [6]

- **ERC-1822**, called the **Universal Upgradeable Proxy Standard (UUPS)**, establishes a standard for proxy contracts that ensures universal compatibility with all types of contracts, eliminating problems of incompatibility between the proxy and business logic contracts. This is achieved through the use of a unique storage location in the proxy contract, dedicated to storing the Logic Contract address. A compatibility verification mechanism ensures that updates are successfully performed, allowing unlimited changes or according to custom logic. ERC-1822 also introduces an innovative method for selecting among multiple constructors while keeping the ability to verify bytecode unaffected. [4]

3.3 Redactable Distributed Ledger

This section focuses on an **academic proposal** that presents a Redactable Blockchain model that we thought was interesting to present. [7]
Redactable distributed ledgers are useful for some Permissioned Distributed Ledger based applications such as identity management, smart contracts, and data sharing.
They offer a flexible solution for modifying smart contracts **post-deployment**, enabling developers to address and rectify design flaws or implement upgrades to existing on-chain smart contracts:

- **Smart Contract Flaw Fixing**: Although a smart contract needs to be carefully designed and thoroughly tested before it can be deployed to distributed ledgers, there may be cases where a deployed smart contract is found to have flaws. To fix such flaws, redaction operations can be used to change the content of the deployed smart contract.

- **Smart Contract Upgrading**: After a smart contact is deployed to distributed ledgers, there may be a need to introduce new functions to the smart contract. Instead of creating a new smart contract, redaction operations can be used to upgrade the existing smart contact to include the new functionalities.

3.3.1 Example - TCH-based Redactable Blockchains

The Trapdoor-Controlled Hash (TCH) mechanism employs a combination of a **Trapdoor Key (TK)** and a **Public Key (PK)** to engineer a hash collision between two distinct messages—namely, an original message *m1* and a subsequent message *m2*.
This process is delineated through two primary operations:

1. Hash(m1, PK, OldPublicParameters) = Hash(m2, PK, NewPublicParameters)

2. NewPublicParameters = Function(TK, m2, PK, OldPublicParameters)

Where Hash() is the function that calculates the hash value of a given message and Function() is another function that calculates NewPublicParamters using the TK, the PK, and OldPublicParameters. It should be computationally challenging for any party to derive NewPublicParameters by brute-force approaches based on Function() without knowing the TK; otherwise, PDL loses its immutability property.
Essentially, there are two steps in TCH to introduce a hash collision:

- **Step 1**: find NewPublicParameters for the new message m2 according to equation 2 using the TK;
- **Step 2**: calculate the hash value of the message m2 using Hash(m2, PK, NewPublicParameters), which will be equal to the hash value of the old message m1 (Hash(m1, PK, OldPublicParameters)), according to equation 1.

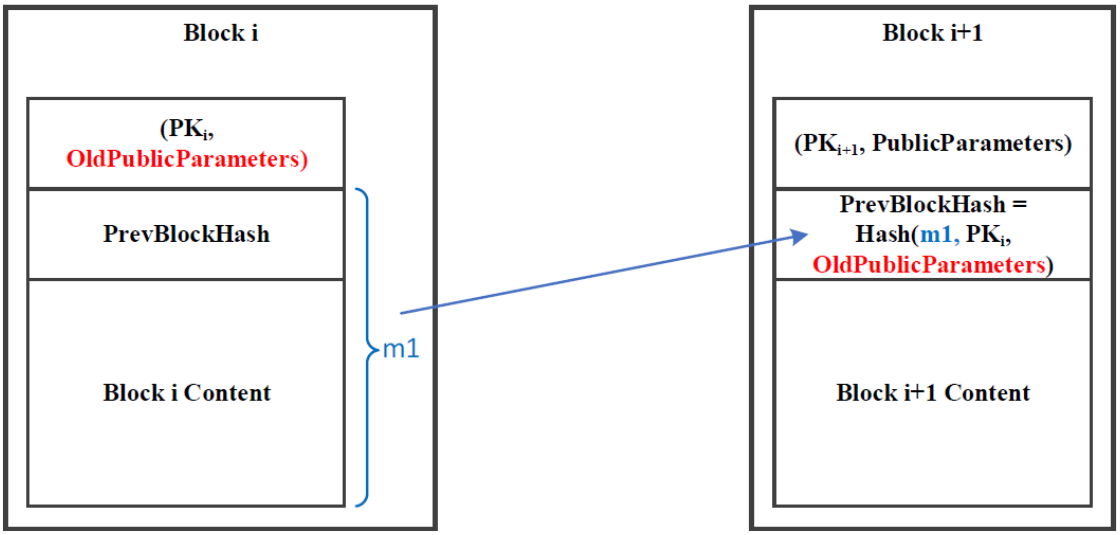


Figure 2: Blockchain before a Redaction Operation

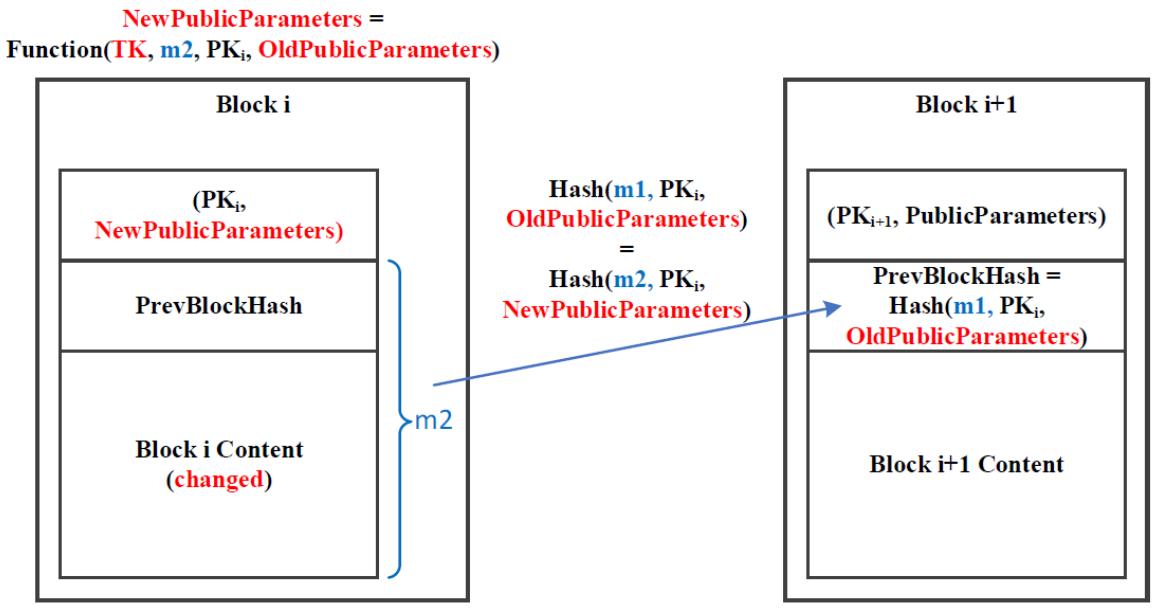


Figure 3: Blockchain after a Redaction Operation

4 Solution

To gain a clearer insight into our solution, it's essential to begin with an overview of our methodology and the systematic approach we employed.

4.1 Process & Methodology

4.1.1 Problem Validation Phase

The initial phase of our project was crucial, as it laid the foundation for all subsequent work. This phase involved a meticulous process of defining the problem at hand, ensuring that we had a clear and precise understanding of the issues we intended to address. We devoted significant effort to outlining the scope of the problem, identifying its key components and potential impacts on stakeholders. In addition, this phase included a comprehensive feasibility study to assess whether moving forward with our project was practical and viable.

4.1.2 Research phase

During the second phase of our project, we proceeded to critically select the solutions that emerged from the state-of-the-art analysis. Importantly, not all pre-existing solutions were included; we chose to focus exclusively on those considered most promising and viable by both ourselves and the community of experts in the field. This filtering process allowed us to identify the most innovative and effective options, ensuring that the solutions tested were those with the greatest potential for success and impact.

4.1.3 Developing & Testing phase

This phase focused on the design and testing of several smart contracts in Solidity using the Foundry platform. The objective was twofold: first, to analyze the advantages and limitations of already available solutions, mainly those offered by OpenZeppelin; and second, to validate the pros and cons of well-known solutions. The results obtained helped to improve the "proposed solution," reinforcing its strengths and identifying any weaknesses.

Use cases

- Bug Correction
- Edit, delete the value of a variable or add a new one
- Edit, delete a feature (method) or add a new one

These use cases are just a generalization of what happens in the real world. They also do not take into account the continuous and rapid development of WEB3 technology. Let us now look at some real scenarios associated with the generic ones listed above to validate the actual need for this more flexible type of contract.

- *Interest Rate Updates:* A financial institution might use a smart contract to manage loans or investments. If interest rates change, the contract could be updated to reflect the new rates, thereby keeping loans or investments in line with current market conditions;
- *Modification of Set Price:* A smart contract used for selling goods or services may include a set price. If the involved parties agree on a price change, the contract could be updated to reflect the new price. This can be particularly useful in long-term supply contracts where prices may fluctuate over time;

- *Bug Fixes in Code:* Smart contracts, like any other software, can contain errors. An upgradable smart contract can allow developers to fix critical bugs or security vulnerabilities without having to interrupt or migrate the existing contract. This can be crucial for ensuring the security and reliability of the contract over time;
- *Adding New Features:* Upgrades don't have to be just bug fixes or data updates. They can also involve adding new features to the contract, allowing the involved parties to take advantage of new opportunities or changes in market needs;
- *Governance Rule Management:* Smart contracts used to manage decentralized organizations (DAOs) or DeFi protocols may require updates to modify governance rules, such as voting procedures or the distribution of ownership shares. Upgrades can be used to improve protocol governance and respond to community needs;
- *Integration of New Standards or Protocols:* With the evolution of the blockchain landscape, new standards or protocols may emerge that offer improved advantages over existing versions. Smart contract upgrades allow for integrating these new standards or protocols without having to completely rewrite the existing contract;
- *Adaptation to Regulatory Changes:* Regulations can change over time, and smart contracts need to be able to adapt to such changes to remain compliant with local laws and regulations. Upgrades can be used to incorporate regulatory changes into contracts, ensuring ongoing compliance.
- *Improvement of Performance and Efficiency:* With advancing technology and evolving best development practices, new techniques for improving smart contract performance and efficiency may emerge. Upgrades can be used to implement these new techniques and optimizations without disrupting existing operations.

4.1.4 Foundry

Foundry is a modern and efficient **toolchain for smart contract development** written in Rust which allow Smart Contract development in Solidity. It offers a wide range of tools and features that simplify and speed up the development process, from code writing to compilation, testing and deployment.[10]

At the heart of Foundry is an intuitive command-line interface (CLI) that facilitates the entire development workflow. From creating new projects and writing Solidity code, to testing, deployment, and interaction with smart contracts, everything is manageable directly from the terminal.

Foundry includes an integrated testing framework that helps identify and fix bugs and vulnerabilities in smart contracts before they are deployed to production. This testing process ensures a higher level of security and reliability.

Foundry is compatible with several common development tools such as Truffle and web3.js, ensuring smooth integration with dApps and existing blockchain infrastructure. Its flexibility makes it suitable for different development needs.

Deploying smart contracts with Foundry is 'easy' and efficient. A single command allows contracts to be deployed on different environments, from testnets to mainnets, and on different

Ethereum Virtual Machine (EVM)-compatible blockchains, such as Polygon, Arbitrum, Optimism and Avalanche.

Key benefits of Foundry:

- Focus on user experience: The intuitive interface and comprehensive documentation make Foundry accessible even to beginners.
- Speed and efficiency: Foundry's tools are optimized for a smooth and fast pace of development.
- Flexibility: Compatibility with different technologies and the modular nature of Foundry make it suitable for various use cases.
- Security: Emphasis on testing and integrations facilitates the development of secure and robust smart contracts.

Limitations to keep in mind:

- Evolving documentation: Some features may not have complete and up-to-date documentation.
- Growing ecosystem: Foundry is a relatively new toolchain, so the ecosystem of third-party libraries and community support may still be developing compared to more mature alternatives.

4.1.5 Final results

The adoption of advanced smart contract writing and deployment solutions, centred on OpenZeppelin and Foundry, has brought to light both positive aspects and some challenges to be considered for future developments.

- *Vulnerabilities of proxies:* A focus of the research has been on the potential vulnerabilities introduced by the use of smart contract proxies. Although they provide transparent updateability mechanisms, great care must be taken in their implementation because errors can pose serious exploitation risks.
- *Distributed governance:* A further consideration concerns the approach to governance of OpenZeppelin solutions. Their decentralized nature offers advantages in terms of transparency and participation; however, it can lead to less efficient decision-making and, in some cases, conflicts of interest. More specifically, the governance models provided by OpenZeppelin libraries are complex and consequently fail to address the needs of the primary users, i.e., developers, causing them difficulties.
- *Cost and time efficiency:* The combination of OpenZeppelin and Foundry has proven effective on the cost and time efficiency of development. OpenZeppelin provides reusable and well-tested components, reducing the need to reinvent the wheel and increasing contract security. Foundry, in turn, accelerates the contract development process with intuitive tools for testing and deployment, as well as offering an accessible user experience.
- *Lack of documentation on Foundry:* One challenge that has emerged is related to the documentation of Foundry. As a relatively new toolchain, some of its functionality may not yet be fully documented. This may slow down the learning and troubleshooting process, but it is something that is expected to improve over time as the project grows.

4.2 Solution Proposal

Following previous works, our proposed solution is based on an abstract framework that effectively integrates the functionalities of the proxy agreement, incorporating the essential procedures for maintaining a transparent version control system. This conceptual framework acts as a model, undergoing further refinement through three distinct subclass extensions, each tailored to address a specific use case.

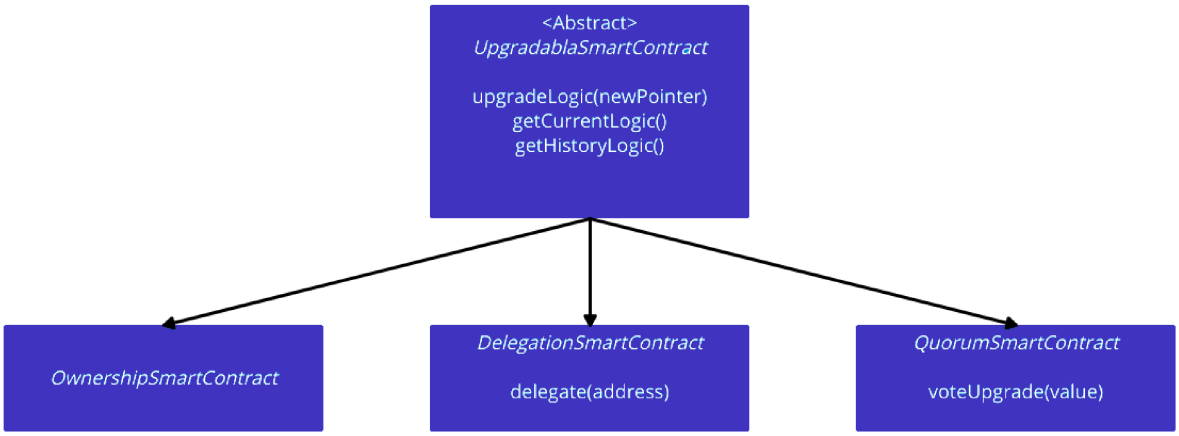
- Ownership Contract: This category is formulated to meet the requirements of a scenario where exclusive update privileges are vested in the contract owner. Essentially, it guarantees that modifications to the underlying logic are only permissible by the proprietor of the contract.
- Delegation Contract: This extension is customized for situations in which the contract owner can grant specific delegates the authority to make updates. This category enables a versatile hierarchy, allowing the owner to delegate update privileges as required.
- Quorum Contract: This category is created to manage situations in which a predetermined array of addresses holds the authority to either approve or reject an update. This set of addresses, forming a quorum, is established and fixed during the initiation of the contract. The quorum agreement essentially facilitates collective decision-making, ensuring that updates are not unilaterally imposed but rather subject to group approval.

4.2.1 Identified improvements

To address the vulnerabilities arising from upgradable smart contracts in OpenZeppelin, we propose the development of a comprehensive library that aims to mitigate, if not eliminate, errors during the web 3 programmers' development phase. This proposed solution encompasses several key points:

- **Enhanced Security Measures:** The library will implement advanced security measures to circumvent vulnerabilities associated with upgradable smart contracts. This includes rigorous testing protocols, code audits, and the integration of automated tools to identify and rectify potential weaknesses. By prioritizing security from the outset, we aim to significantly reduce the likelihood of errors during the development process.
- **Transparent Governance Structure:** Clarity in governance is crucial for the effective management of smart contract upgrades. Our solution will include a well-defined governance framework, clearly outlining the roles and responsibilities of each stakeholder. Additionally, we will introduce mechanisms to make the upgrade process more transparent, ensuring that stakeholders understand how, when, and by whom upgrades can be executed. This transparency will instill confidence in users and developers alike.
- **User-Friendly Upgrade Processes:** To simplify the upgrade process, the proposed library will prioritize user-friendliness. This involves creating intuitive interfaces and documentation to guide developers through the upgrade process seamlessly. By enhancing accessibility and providing comprehensive documentation, we aim to empower developers and reduce the likelihood of errors caused by misunderstandings or misconfigurations during upgrades.
- **Code Optimization Strategies:** Recognizing the importance of efficient code, our solution will focus on optimizing the smart contract codebase. This involves streamlining

and improving the efficiency of the code to enhance overall performance. Code optimization not only contributes to a more robust and secure system but also facilitates easier maintenance and reduces the risk of potential vulnerabilities.



4.2.2 Positioning & Impact

Our solution aligns with a burgeoning domain that has already captured the attention and investment of the European Union, placing it at the forefront of technological innovation. As of 2018, the global community of Blockchain developers stood at a modest 105,000 [1], a fraction of the 18 million-strong software developer workforce worldwide. However, the landscape has been rapidly evolving. By the close of 2021, during the pinnacle of Bitcoin and other cryptocurrency valuations, a staggering 18,500[20] new developers were joining the ranks each month to contribute to open-source Web3 projects. This surge is indicative of the growing interest and engagement in the blockchain space, as revealed in a report by Electric Capital. Notably, the number of Solidity developers, a programming language commonly used for smart contracts, was estimated to be around 200,000 in 2020, underscoring the swift expansion of blockchain expertise. With the increasing prominence of blockchain technologies, as outlined in the earlier sections of this document, these figures are poised to experience sustained growth.

Expanding our focus to the indirect impact, the presence of 63 million smart contracts[17]exclusively on the Ethereum blockchain exemplifies how this decentralized network has efficiently facilitated agreements, binding a minimum of 126 million parties. This underscores a twofold impact on contractual interactions and digital transactions. The expansive network of smart contracts serves as a testament to the profound influence of blockchain technologies, reshaping the global landscape of trust and transactions. With the ongoing proliferation of blockchain adoption, these indirect impacts are anticipated to reverberate across diverse industries and sectors.

5 Conclusions & Future developments

Based on the results of our research and testing, we are aware that our theoretical solution represents only a starting point. The first necessary step will be the development of our proposed library. In addition, working with *Professor Liquori*, we have explored the opportunity to implement a low-level modification on the EVM. This modification would introduce the concept of **Dynamic Binding**, or a variant of it, with the ambition of simplifying the development of Smart Contracts. Specifically, the innovation aims to allow developers to apply lower adjustments rather than having to rewrite and redeploy the entire Smart Contract, thereby optimizing execution costs and reducing development complexity.

These reflections are based on theories that are, from our perspective, fascinating for the enhancement of Upgradeable Smart Contracts, a field that Giovanni Rigotti will explore further in his upcoming internship at **INRIA**. This internship is part of broad research contexts, including initiatives related to data spaces in the **Digital Europe program**, Horizon 2020 and Horizon Europe, as well as collaboration with **ETSI ESI** and other relevant technical committees at the European and international level, involving companies such as *Nokia, Huawei and Infocert*. More precisely, developing a scoping study analysing the requirements on Smart Contracts from the Data Act and **eIDAS2** proposals and identifying standardisation requirements for smart contracts in data sharing applications.

It is important, however, to anticipate the challenges this proposal may face, including Ethereum's acceptance of EVM modification and the implications on research speed due to the preference for Dynamic Binding over Static Binding.

In our study, we observed that Web3 programming languages and related technologies are still in an early stage of development. We envision that the coming years will see intensified efforts to improve and refine these technologies, with the goal of bridging the gaps between current capabilities and emerging needs in real-world application contexts. This progress will be crucial to harnessing the full potential of Web3 in transforming industries such as fintech and DeFi, making them more accessible, secure, and efficient.

References

[1] *Blockchain developers number*. URL: https://paybis.com/blog/how-many-blockchain-developers-are-there/. (accessed: 10.02.2024).

[2] *Blockchain market size*. URL: https://www.statista.com/statistics/1015362/worldwide-blockchain-technology-market-size/. (accessed: 15.12.2023).

[3] *ERC-1167: Minimal Proxy Contract*. URL: https://eips.ethereum.org/EIPS/eip-1167. (accessed: 08.01.2024).

[4] *ERC-1822: Universal Upgradable Proxy Standard*. URL: https://eips.ethereum.org/EIPS/eip-1822. (accessed: 01.02.2024).

[5] *ERC-1967: Proxy Storage Slots*. URL: https://eips.ethereum.org/EIPS/eip-1967. (accessed: 05.01.2024).

[6] *ERC-2535: Diamond Multi-Faced Proxy*. URL: https://eips.ethereum.org/EIPS/eip-2535. (accessed: 05.01.2024).

[7] *ETSI Redactable Distributed Ledger*. URL: https://www.etsi.org/deliver/etsi_gr/PDL/001_099/018/01.01.01_60/gr_PDL018v010101p.pdf. (accessed: 20.02.2024).

[8] *Financial losses caused by attacks targeting SCs*. URL: https://arxiv.org/pdf/2304.02981.pdf. (accessed: 15.12.2023).

[9] *Foundry Book*. URL: https://book.getfoundry.sh/. (accessed: 16.01.2024).

[10] *Foundry book*. URL: https://book.getfoundry.sh/. (accessed: 15.01.2024).

[11] *Foundry Cheatsheet*. URL: https://github.com/dabit3/foundry-cheatsheet. (accessed: 18.01.2024).

[12] *Foundry Workshop*. URL: https://github.com/dabit3/foundry-workshop?search=1. (accessed: 18.01.2024).

[13] *Investigation of the number of vulnerable contracts*. URL: https://coinjournal.net/it/notizie/ethereum-trovati-34-200-smart-contracts-vulnerabili-agli-attacchi/. (accessed: 29.11.2023).

[14] *OpenZeppelin - Writing Upgradeable Contracts*. URL: https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable. (accessed: 16.01.2024).

[15] *Proxy Upgrade Pattern*. URL: https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies. (accessed: 05.01.2024).

[16] *Setup Foundry for solidity*. URL: https://www.youtube.com/watch?v=s7vpfH6spcM. (accessed: 15.01.2024).

[17] *Smart Contract Deployment Statistics*. URL: https://dune.com/pcaversaccio/smart-contract-deployment-statistics. (accessed: 28.02.2024).

[18] *Smart Contract inheritance*. URL: https://docs.alchemy.com/docs/smart-contract-inheritance. (accessed: 19.01.2024).

[19] *The State of Smart Contract Upgrades*. URL: https://blog.openzeppelin.com/the-state-of-smart-contract-upgrades. (accessed: 15.01.2024).

[20] *Web3 Growth Stymied*. URL: https://www.forbes.com/sites/ninabambysheva/2022/08/29/web3-growth-stymied-by-scarcity-of-programmers/. (accessed: 05.02.2024).

[21] *Web3 Tutorial*. URL: https://dev.to/yakult/tutorial-write-upgradeable-smart-contract-proxy-contract-with-openzeppelin-1916. (accessed: 15.01.2024).

List of Figures

A Foundry: Documentation

A documentation/guide for the installation and first use of Foundry;

A.1 FOUNDRY BASICS

It is based on *3 Command Line Interface (CLI) tools*:

- **ANVIL:**
  Is a local Ethereum node, similar to Ganache. Useful for testing your contracts.
- **FORGE:**
  Used to test, build and deploy your smart contracts.
- **CAST:**
  Allows you to interact with EVM smart contracts, send transactions and read data from the network.

A.2 STEPS

Get Starting with Foundry

1. Install WSL - (Ubuntu) [16]
2. Download foundry on the "new ubuntu terminal"
   Command: `curl -L https://foundry.paradigm.xyz` bash—
3. Install Foundryup:
   Command: `foundryup`
4. Check version:
   Command: `forge -version`
   Command: `cast -version`
   Command: `anvil -version`
5. Move to the location of the new project:
   Command: `[Linux basics: cd ls pwd]`
6. Create empty/new Foundry project:
   Command: `forge init <Project name>`

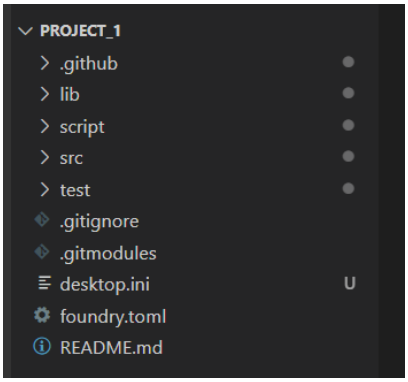


Figure 4: Foundry - Starting Solution Structure

7. Open a WSL Terminal in VS Code:

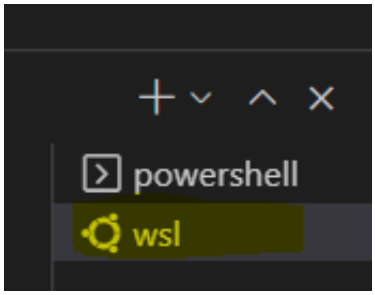


Figure 5: VSC - WSL Terminal

8. Building the project:
   Command: `forge build`

9. Testing the project:
   Command: `forge test`

   - To view logs text add the emit log command as follows in the *'t.sol'* file:
     Code: `emit log("Hello World")`

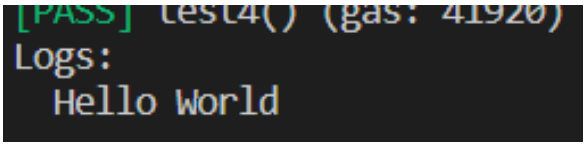


Figure 6: Forge test log

Deploy the contract on a test network/local node

1. Run the local node on a different WSL Terminal:
   Command: `anvil`
   This will start a local network and spin up 10 accounts and private keys and log them out to the console. Once the network is running, we can use forge to deploy the contract to the network through the scripts.

2. Next, set the *PRIVATE_KEY* variable by using one of the private keys given to you by Anvil:
   Command: `export PRIVATE_KEY=<your-private-key>`

3. To deploy, run this script:
   Command: `forge script script/HelloWorld.s.sol:ContractScript --fork-url http://localhost:8545 --private-key $PRIVATE_KEY --broadcast`
   Once the contract is deployed, the contract address will be logged out to your terminal.

4. Set the CONTRACT_ADDRESS variable in your terminal:
   Command: `export CONTRACT_ADDRESS=<your-contract-address>`

We can then use *cast* to interact with it.

1. For read operations, we can use cast call:
   Command: `cast call $CONTRACT_ADDRESS "greet()(string)"`
   Where greet() is a method defined into the Smart Contract that returns a string (without paying fees as a public view)

2. For transactions, we can use cast send, passing in a private key and any arguments:
   Command: `cast send $CONTRACT_ADDRESS "updateGreeting(string)" "My new greeting" --private-key $PRIVATE_KEY`

3. To test that the greeting has been updated, run the call command again:
   Command: `cast call $CONTRACT_ADDRESS "greet()(string)"`

Installing packages

1. To install OpenZeppelin Contracts:
   Command: `forge install OpenZeppelin/openzeppelin-contracts`

2. To see al the names/path mappings:
   Command: `forge remappings`

3. To modify/add remappings we have to create a file named remappings.txt into the "parent" folder and add the new remapping:
   e.g.: `@openzeppelin/=lib/openzeppelin-contracts/`
   This will allow us to easily import with the following syntax:
   `import "@openzeppelin/contracts/token/ERC20/ERC20.sol";`

4. Additional steps into Foundry Workshop in "SUPPORT MATERIAL" section.

Proxy contract guides:
Web3 Tutorial [21]
Writing Upgradeable Contracts [14]

SUPPORT MATERIAL:
Foundry book [9]
Foundry Cheatsheet [11]
Foundry Workshop (tutorial to test all the basics foundry functions) [12]
Smart Contract inheritance [18]

A.3 Code: Explanation & Testing

A brief explanation of the code and the results with attached use cases/topics tested in the related solution. Below is the image with the solutions developed in Foundry to learn how to use the platform and to test Smart Contracts on Ethereum.

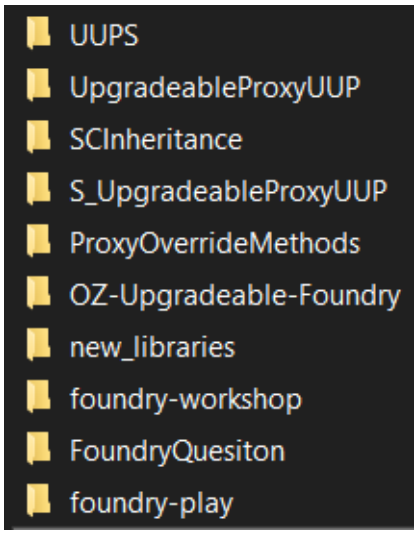


Figure 7: Foundry solutions

A.4 Solution: foundry-workshop

This solution consists of two distinct contracts:

- *HelloWorld Contract:* This contract allows for the setting and updating of a greeting message. It features functions to retrieve the current greeting and update it with a new one, along with a version tracking mechanism. Users can interact with the contract to manage the displayed greeting message.

```
contract HelloWorld {
  string private greeting;
  uint public version = 0;

  constructor (string memory _greeting) {
    greeting = _greeting;
  }

  function greet() public view returns(string memory) {
    return greeting;
  }

  function updateGreeting(string memory _greeting) public
      {
    version += 1;
    greeting = _greeting;
  }
}
```

- *DevconPanda Contract:* This contract is an ERC721-compliant non-fungible token (NFT) contract named DevconPanda. It inherits from ERC721URIStorage, enabling it to store metadata associated with each token. The contract includes a minting function to create new tokens with unique identifiers and associated metadata URI, providing a basis for creating and managing NFTs within the Ethereum ecosystem.

```
contract DevconPanda is ERC721URIStorage {
    // using Counters for Counters.Counter;
    // Counters.Counter private _tokenIds;
    uint256 private _tokenIds = 0;

    constructor() ERC721("DevconPanda", "DCP") {}

    function mint(address user, string memory tokenURI)
       public returns (uint256) {
        uint256 newItemId = _tokenIds;  // Counter.
           current()
        _mint(user, newItemId);
        _setTokenURI(newItemId, tokenURI);

        _tokenIds = _tokenIds + 1; // Counter.increment()
        return newItemId;
    }
}
```

Tested: Foundry and SC developing Learning purpose, OpenZeppelin Basics ERC import and usage.

A.5 Solution: foundry-play

This solution contains a smart contract, *"StakeContract"*, that facilitates staking ERC20 tokens by users. It maintains a mapping of user balances and utilizes the *transferFrom* function from the *IERC20* interface for token transfers, ensuring secure and efficient staking operations. Additionally, it handles transfer failures through a custom error mechanism, enhancing robustness and reliability in token staking processes.
Tested: TOKENIZATION, ERROR HANDLING

A.6 Solution: FoundryQuestion

In fact, this solution is not currently compileable because it contains only a few files with Smart Contract that generated an unknown error during testing and development. This solution was created for the purpose of collaborating with an experienced developer in Solidity/Foundry with whom we were fortunate enough to attend a class during the P2P course taught by Professor Liquori. With an active chat with the expert Amas Louis, we therefore came to the conclusion that we needed to update the support libraries. He also suggested a different strategy for us to develop more comfortably with Foundry.
bl **Tested:** Learning Purposes, Error Handling, Dependencies update

A.7 Solution: OZ-Upgradeable-Foundry

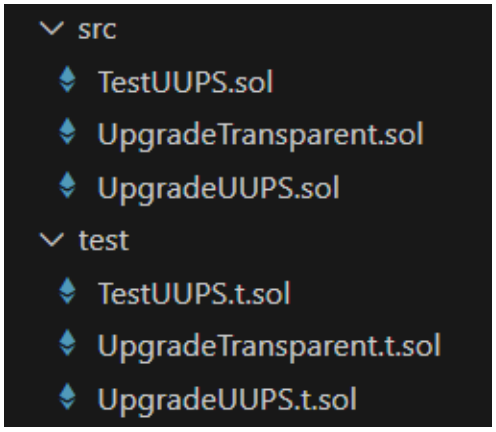


Figure 8: First OpenZeppelin Upgradable ERC Test

These contracts are designed to showcase various upgradeability patterns and functionalities within the Ethereum ecosystem. Users can explore and interact with these contracts to understand how upgradeable contracts are implemented and managed on the Ethereum blockchain.

The following test script shows how these solutions work and gave us an idea of which was best for us in terms of ease of use/development and performance:

```
    contract _Test is PRBTest {
        // State variables for the contract.
        MyContract implementationV1;
        UUPSProxy proxy;
        MyContract wrappedProxyV1;
        MyContractV2 wrappedProxyV2;

        // setUp function to initialize the state of the contract
            .
        function setUp() public {
            // Create a new MyContract contract.
            implementationV1 = new MyContract();

            // Deploy a new UUPSProxy contract and point it to
               the MyContract implementation.
            proxy = new UUPSProxy(address(implementationV1), "");

            // Wrap the proxy contract in the ABI of the
               MyContract contract to support easier calls.
            wrappedProxyV1 = MyContract(address(proxy));

            // Initialize the wrapped proxy contract with a value
                of 100.
            wrappedProxyV1.initialize("v1");
        }
```

```
        // Test function to check if the contract can be
           initialized.
        function testCanInitialize() public {
            // Assert that the value of 'x' in the wrapped proxy
               contract is 100.
            assertEq(wrappedProxyV1.version(), "v1");
        }

        // Test function to check if the contract can be upgraded
           .
        function testCanUpgrade() public {
            // Create a new MyContractV2 contract.
            MyContractV2 implementationV2 = new MyContractV2();

            // Upgrade the wrapped proxy contract to the new
               implementation.
            wrappedProxyV1.upgradeTo(address(implementationV2));

            // Re-wrap the proxy contract in the ABI of the
               MyContractV2 contract.
            wrappedProxyV2 = MyContractV2(address(proxy));

            // Assert that the value of 'x' in the re-wrapped
               proxy contract is still 100.
            assertEq(wrappedProxyV2.version(), "v1");

            // Set the value of 'y' in the re-wrapped proxy
               contract to 200.
            wrappedProxyV2.setAge("a1");

            // Assert that the value of 'y' in the re-wrapped
               proxy contract is now 200.
            assertEq(wrappedProxyV2.age(), "a1");
        }
    }
```

A.8 Solutions:

- UpgradeableProxyUUP
- S_UpgradeableProxyUUP
- UUPS

These three solutions contain variations of the same contracts with different testing purposes in order to go and evaluate precisely all the use cases defined in the report.
Tested: ERC1967Proxy Standard, Upgrade a version of a contract on a proxy, Add/Change a variable, Read/Write from/to a file in Foundry.

A.9 Solution: SCInheritance

This repository consists of contracts representing different entities within a family structure:

- *Human.sol:* Defines the Human contract, featuring a state variable for the name, an event for name changes, and a modifier restricting certain functions to the owner. It includes functions for changing the name and viewing the entity type, with a default description of "I'm an Human".
- *Parent.sol:* Inherits from Human, defining the Parent contract. It initializes the number of children and overrides the view function to return "I'm a Parent".
- *Son.sol:* Inherits from Parent, defining the Son contract. It overrides the view function to return "I'm a Son".
- *Daughter.sol:* Inherits from Parent, defining the Daughter contract. It overrides the view function to return "I'm a Daughter".

Figure 9: Inheritance Test Solution

These contracts illustrate inheritance and polymorphism, showcasing the family hierarchy within a Solidity smart contract context. The scope of this solution was to assess the known "problem" of the inheritance behaviour of the Smart Contracts as the management through *Inlining/Static Binding* despite of *Dynamic Bindings*.
Tested: Inheritance in general, SC on ETH behaviour (Static vs Dynamic Binding)

A.10 Solution: new_libraries

Despite the misleading name this solution was intended to *reproduce the behaviour of a proxy* (ERC1967Proxy by OpenZeppelin) developed without the aid of the library. Below is the code.

```
// Contract for Proxy based on ERC1967Proxy
contract Proxy {
    // Address of the current implementation
    address private _implementation;

    // Data structure to hold the slots
    mapping(bytes32 => bytes32) private _slots;

    constructor(address _initialImplementation) {
        _implementation = _initialImplementation;
```

```
    }

    fallback() external payable {
        _fallback();
    }

    receive() external payable {
        _fallback();
    }

    function _fallback() private {
        bytes32 slot = keccak256(abi.encodePacked("implementation
           "));
        assembly {
            let _target := sload(slot)
            calldatacopy(0x0, 0x0, calldatasize())
            let result := delegatecall(gas(), _target, 0x0,
               calldatasize(), 0, 0)
            returndatacopy(0, 0, returndatasize())
            switch result
            case 0 {
                revert(0, returndatasize())
            }
            default {
                return(0, returndatasize())
            }
        }
    }

    function upgradeTo(address newImplementation) external {
        _implementation = newImplementation;
    }

    function implementation() external view returns (address) {
        return _implementation;
    }
}
```

We decided to do this test to understand firsthand how the standard provided by *OpenZeppelin* works (being able to debug the code outside the library).

A.11 Solution: ProxyOverrideMethods

In this last solution we finally combined all the skills we had acquired to simulate real bug cases, use inheritance for upgradable smart contract management, and make the ovveride of a bug-containing method (in this case obviously naive).

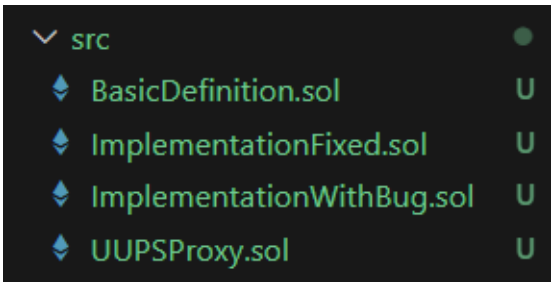


Figure 10: Proxy Override Methods Solution

In the test we first created a proxy pointing to a bugged implementation. We only subsequently deployed and version containing the fix that thanks to inheritance allowed us to do ovveriding of the buggy method.

```
Compiler run successful!
giova@PC-GIOVA:/mnt/c/Users/rigot/Desktop/Codes/ProxyOverrideMethods$ forge test -vv
[⠒] Compiling...
No files changed, compilation skipped

Ran 2 tests for test/TestUUPS.t.sol:_Test
[PASS] testImplemetationWithBug() (gas: 18953)
Logs:
  sum would be increased by 1 in this version instead of:  30
  SUM RETURN 31 INSTEAD OF 30 --> BUGGED

[PASS] testUpgradeAndImplementationFixed() (gas: 995670)
Logs:
  sum would be correct in this version
  SUM RETURN 30 AS EXPECTED --> FIXED

Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 1.50ms

Ran 1 test suite in 1.50ms: 2 tests passed, 0 failed, 0 skipped (2 total tests)
```

Figure 11: Proxy Override Methods Test

With this solution, we then **tested and verified** all the use cases defined in the report as: Bug Correction, Add,Delete,Modify Variables, Add,Delete,Modify Features/Methods. This is how we were able to find limitations and benefits of existing solutions proposed by OpenZeppelin and then apply them to our proposed solution.