THE JBBA

PEER REVIEWED RESEARCH

OPEN ACCESS
ISSN Online: 2516-3957
ISSN Print: 2516-3949
https://doi.org/10.31585/jbba-6-1-(3)2023

A Review of Upgradeable Smart Contract Patterns based on OpenZeppelin Technique

Shaima AL Amri, Leonardo Aniello, Vladimiro Sassone
School of Electronics and Computer Science, University of Southampton, UK

Correspondence: saaa1n18@soton.ac.uk/shaima.alamri@hotmail.com
Received: 30 December 2022 **Accepted**: 30 January 2023 **Published**: 20 March 2023

Abstract
The Ethereum blockchain is one of the main public platforms to run smart contracts and enable decentralised applications. Since data stored in a blockchain is considered immutable, smart contracts deployed in Ethereum are regarded as tamper-proof and therefore offer strong protection against attacks aiming at tinkering with the execution flow of an application. Yet, like any other software, a smart contract needs to be maintained over time to fix bugs or add new features. Deploying every updated version as a brand-new smart contract in Ethereum leads to problems such as migrating the contract state from the old version and enabling clients to point to the new version in a timely fashion. The OpenZeppelin framework addresses this limitation by providing libraries that enable the deployment of upgradeable smart contracts. This is achieved by relying on proxies that act as intermediaries between clients and smart contracts, allowing the latter to be updated transparently. In this paper, we present the upgradeable smart contract patterns supported by OpenZeppelin and compare them in terms of security, cost, and performance. To show this paradigm's prevalence in Ethereum, we also analyse the usage of OpenZeppelin Upgradeable smart contracts over the last four years.

Keywords*: Smart Contract, Immutability, Proxy, Upgradeable, Patterns, OpenZeppelin*

JEL Classifications: *C80 and C89*

1 Introduction

The upgraded smart contract (USC), designed with the OpenZeppelin technique, is used to reserve the contract state and redirect calls to the implementation contract (IC). In addition, it upgrades the IC address when it requires a new version. In this technique's design, the proxy stores the state of the IC and maps any new version to the same state [12]. The upgraded technique is not preventing the known attacks against smart contracts. The purpose of designing the USC is to ensure the state cannot be lost and will be mapped to a new version of IC.

OpenZeppelin is a widely used library for writing secure smart contracts, and it provides several patterns for implementing upgradability [6]. These patterns include TransparentProxy, ERC1967, UUPSUpgradeable, and Beacon [6]. The research reviews the OpenZeppelin Upgradeable patterns to understand how they vary based on gas consumption, security, and performance.

The OpenZeppelin technique of upgrading smart contracts was invented in 2017. This technique motivates us to analyse the verified smart contract in Etherscan and check whether this technique has grown within the last four years.

The paper is organised as follows. Section 2 introduces Ethereum and smart contracts, followed by the OpenZeppelin Upgradeable technique. Section 3 presents the research methods, while Section 4 presents the analysis of upgraded patterns and compares them based on gas consumption, security, and performance. Section 5 shows the analysis result of the usage of upgradeable patterns over the last four years. Finally, Section 6 summarises our review and outlines our future direction.

2 Background

2.1 Ethereum and Smart Contract

Ethereum platform is one of the public blockchain platforms introduced by Vitalik Buterin [1] to override the limitations of Bitcoin's scripting language. Unlike Bitcoins, Ethereum innovated to support all loops as it is full of Turing completeness [10]. Moreover, the Ethereum platform is the most popular for deploying contract-based applications in several contexts, including financial services, insurance, education, healthcare, and cryptocurrencies.

A smart contract is designed to verify and enforce legal contract negotiation. Once it is deployed in the blockchain, no one can change the code [2]. It is executed and verified by Ethereum Virtual Machine Environment (EVM), built within all decentralised nodes in the Ethereum blockchain. The immutability characteristic of the smart contract made it a trusted application.

2.2 OpenZeppelin Upgradeable Technique

OpenZeppelin is a popular open-source framework for building secure, modular, and reusable smart contracts on the Ethereum platform [11]. The most important approach is implementing upgradeable smart contracts on the Ethereum platform. Their approach utilises a proxy, allowing developers to deploy a contract as an intermediary between a user and the implementation contract [12]. The proxy will store the state of the implementation contract to reserve the state when upgrading the IC to a new version. The upgrade process is done by invoking the **UpgradeTo( )** function by the EOA, but if the IC had the same function, this would cause a problem known as function clashing [7]. This issue can occur because each function in a smart contract is identified by four-byte at the bytecode level [7] and can not be tracked by the Solidity compiler because we have two different contracts: proxy and IC. To solve this issue, OpenZeppelin has designed a Transparent Proxy pattern [6] with a specific ProxyAdmin contract which plays the admin role. In addition, the invocation of any function will be delegated to the suitable contract according to the caller's address [7]. The ProxyAdmin will be the only one that have the right to invoke the **UpgradeTo( )** function of the proxy.

3 Methodology

The methodology proposed in this research to analyse the upgradeable patterns involves the following steps:

1. **Data Collection:** We obtained the smart contract address and creation timestamp from the Kaggle [4] dataset, which contains live data of the Ethereum blockchain. Then, the data was categorised based on the creation timestamp from 2019 to 2022.
2. **Source Code Crawling**: We crawled source codes using the tool provided by [5] and edited them according to our needs by using the contract addresses from the previous step as input for our script. We were able to crawl different contracts categorised by years.
3. **Data Analysis**: We classified the dataset according to OpenZeppelin Upgradeable patterns (e.g., TransparentProxy, UUPSUpgradeable, ERC1967, and Beacon). We used Python panda's library to classify and calculate summary statistics for each category. The result of this step will be discussed and is present in Section 5.

4 Analysis of OpenZeppelin Upgradeable Patterns

4.1 Upgraded Patterns

The upgradability of a smart contract is done by creating a new version of the deployed contract. The new version is designed to overcome the issues which exist in the old version or by adding new features according to the business needs. The state of the old version will be mapped to the new version. However, the upgradability cannot prevent the smart contract from potential adversaries such as a reentrancy attack.

The OpenZeppelin technique has proposed different patterns. Each pattern is linked with a different contract to achieve its purpose. The **Proxy contract** is represented as the central core of the upgradeable contract with all patterns because it is the core of implementing the delegation functionality [6]. The remaining patterns are designed with different functionalities.

4.1.1 Transparent Proxy

The Transparent Proxy contract is designed to avoid the function clashing and ensure that only the admin can call the upgrade function. The following contracts are the main contracts for implementing this pattern. Figure 1 presents the **deploying T_x** of this pattern, where it clearly shows how the three contracts are created and attached to it is code and storage. The storage of TransparentUpgradeable Proxy is responsible for managing the state of itself and IC [6]. Moreover, the storage of the IC will be useless as IC is responsible for executing the delegated function and sending the output back to the TransparentUpgradeable contract.

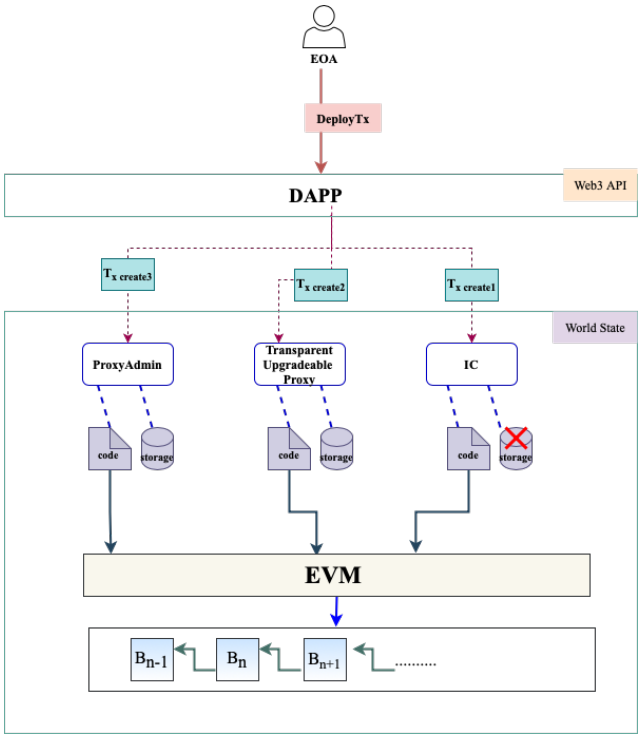

Figure 1. Process of deploying the TransparentProxy pattern

- **TransparentUpgradeableProxy** is designed to manage the calls from the end-user to the IC by checking the caller's identity [8]. In case the caller is the admin, his call will only be delegated to the IC for execution if the caller is an external account.
- **ProxyAdmin** is designed as an admin of TransparentUpgradeableProxy. It only has the right to access the admin functions, which are used for upgrading the proxy or changing the contract owner [8]. For that, the ProxyAdmin is always assigned to a dedicated account.

In case the external accounts invoke a function in IC, the **TransaparentUpgradeable Proxy** checks whether this call is to the admin or IC. The proxy will be delegated the request to the IC if it is valid. The proxy will update the state of IC contract once the request is executed successfully and the output is forwarded to the external accounts. **Figure 2** illustrates how the proxy manged the IC state. The proxy tracks the state of IC because all invocation to IC is only done through the proxy.

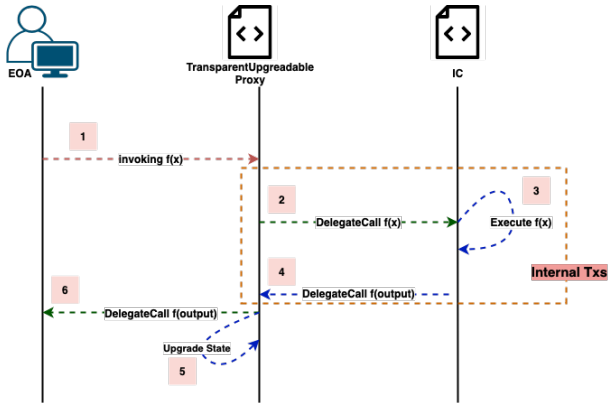

Figure 2. Invoking process of IC functions

Upgrading the IC to a new version requires deploying the new version first and upgrading the address in proxy storage via ProxyAdmin contract. It is clearly shown in **Figure 3** how the upgrade function is executed by the TransparentUpgradeable Proxy once the ProxyAdmin validates the identity of the caller.

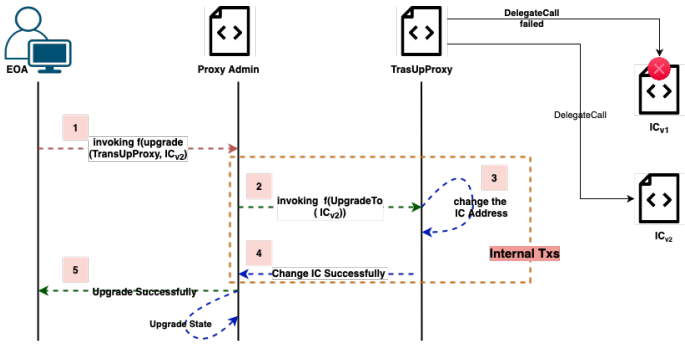

Figure 3. The Upgrade process of Upgrade function

4.1.2 ERC1967

ERC1967Proxy is designed based on EIP1967 [2], which is proposed to overcome any clashes that might occur with the storage layout of IC address. This proxy is not upgradeable by default. It is inherited into the Proxy contract, which was designed as the core for delegating the functions. The function responsible for declaring the storage slot of the IC address showed in **Snapshot 1**. The function is defined as a constructer because it will be executed once the contract is deployed. In addition, the variable of IC address can be changed. Figure 4 illustrates the contract created by deploying the ERC1967 pattern. It shows how the**ERC1967Proxy** is inherited within the proxy as one contract because it helps the proxy guarantee that the **ERC1967Upgrade** is designed along with ERC1967Proxy to "provide the getters and events which emit the upgrade functions of EIP1967 slots" [2].

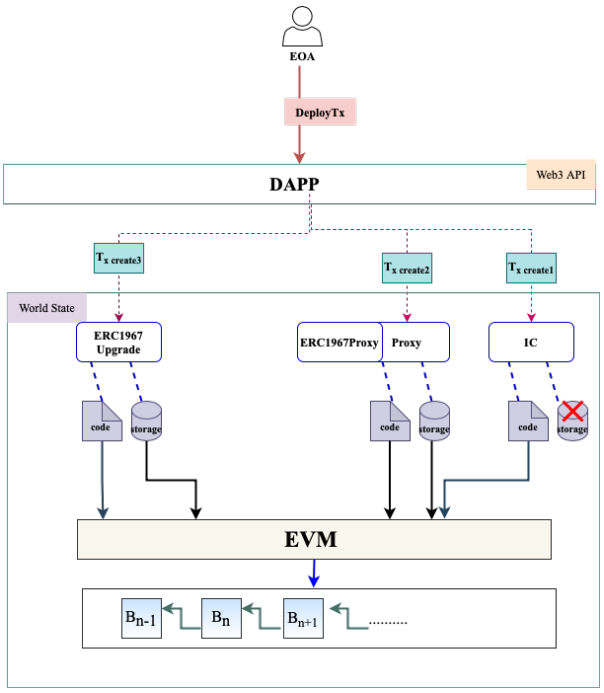

Figure 4. Process of deploying the ERC1967 pattern

The message invocation process using this pattern is similar to the previous pattern in Section 4.1.1. In this pattern, the proxy will delegate the message to IC. Then IC executes the targeted function and returns the output to a proxy. Finally, the proxy will delegate the output to the end-user and update the state of IC.

```
constructor(address _logic, bytes memory _data)
payable { assert(_IMPLEMENTATION_SLOT ==
bytes32(uint256(keccak256("eip1967.proxy.implemen
tation")) - 1));

_upgradeToAndCall(_logic, _data, false);}
```

Snapshot 1. The function of identifying the storage slot of IC address

In the case of invoking the upgrade function, the transactions will be executed differently. The owner initiates the invocation after creating the new contract version and getting IC address. The IC address will be upgraded by the ERC1967Upgrade, where it executes the **_getImplemntation(old_ICaddress)** function followed by calling the **_UpgradeTo(newIC_address)** [6]. The execution of those functions is emitting the old address to the new address, as shown in Figure 5.

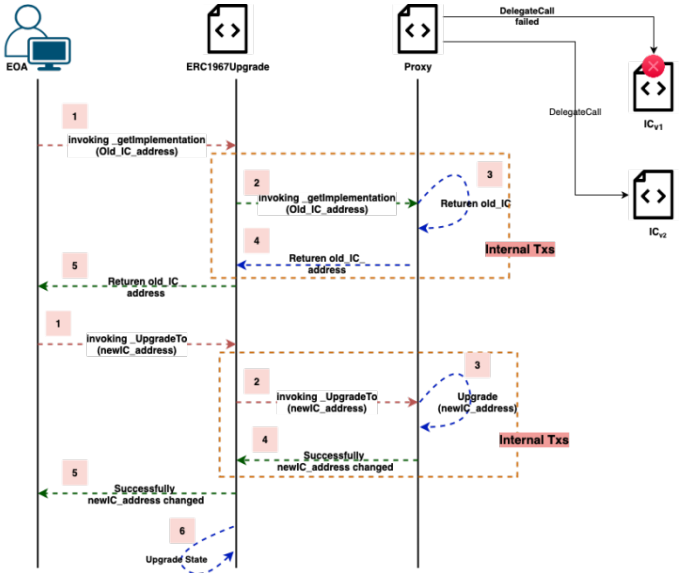

Figure 5. The process of Upgrade using ERC1967 pattern

4.1.3 Beacon

This pattern is designed to call the upgrade function of different **ICs** through multiple proxies. It consists of three contracts, as detailed below.

- **BeaconProxy** is implemented as a proxy designed to retrieve the **IC** addresses for each call initialised by the UpgradeableBeacon contract [6].
- **IBeacon** is designed as the interface of BeaconProxy, as it is responsible for storing the **IC** addresses. The BeaconProxy calls the **implementation()** function [6] and checks whether the return value is a contract or not. Then the return address will be used to delegate the call of the target.
- **UpgradeableBeacon** acts as an admin who has the right to upgrade the **BeaconProxy**. The upgrade process is done by calling the **IBeacon** contract, which holds all IC addresses, and the linked proxies will be upgraded automatically [6].

Figure 6 illustrates the deployment process of Beacon patterns, where four different contracts are created. This pattern is similar to ERC1967 during the upgrade process. Figure 5 as the upgrade IC address to the new version is done through in the IBeacon contract as the BeaconProxy is retrieving the address of IC from IBeacon. However, the interaction between this contract and the end-user is different. In the case of invoking a function in IC while using the Beacon pattern, the call goes through two transactions before being executed by the IC. First, the BeaconProxy will receive the call and retrieve the IC address from the IBeacon to delegate the call. Then, once the BeaconProxy gets the IC address it will delegate the call to the IC for executing the function, as shown in Figure 7.

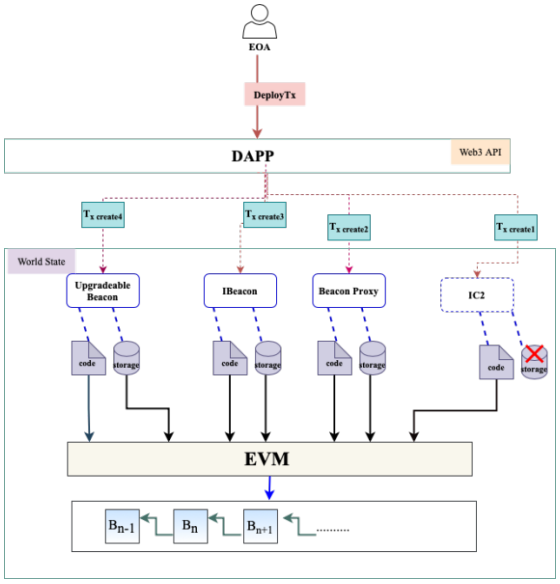

Figure 6. Process of deploying the Beacon pattern

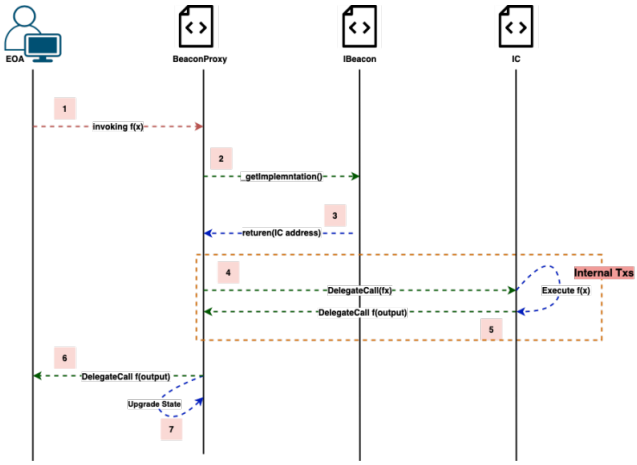

Figure 7. Process of interaction with the Beacon pattern

4.1.4 UUPSUpgradeable

This pattern is considered an upgradability mechanism built within the IC contract. It is implemented by using ERC1967Proxy [6]. In this pattern, the data and contract logic are designed as one contract, as illustrated in Figure 8. The invocation of any function is done directly between the end-user and the contract. However, the upgrade process to a new version will be authorised first by executing the function **_authorisedUpgrade(Owner Address)** [6].

All OpenZeppelin Upgradeable patterns are designed to preserve the immutability of a contract by holding the implementation contract state within a proxy contract. This technique allows the implementation contract to be upgraded without losing the contract state. Some patterns, such as the **TransparentProxy** pattern, are designed to prevent authorisation issues by allowing a fixed admin to upgrade the implementation contract. These patterns can be used to ensure the contract remains upgradeable and secure. For example, the

proxy contract can be deployed using the **TransparentProxy** pattern, which allows the admin to upgrade the implementation contract as needed. In addition, using the **ERC1967** by some patterns is also helpful in avoiding the proxy selector clashing issue.

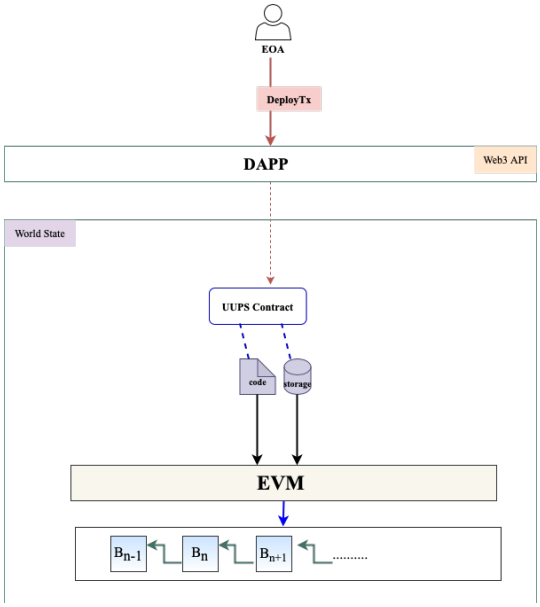

Figure 8. Process of deploying the UUPSUpgradeable pattern

4.2 Comparison of Upgradeable Patterns

4.2.1 Gas Consumption (gc)

The contract transactions executed within the EVM and the cost gas consumption (C_{gc}) for this operation are calculated according to the transaction type. The gas price is changed rapidly according to the network usage,[1] where the amount of consumed gas is a fixed value identified within Appendix G[3]. The (**gc**) of the initial transaction of creating a smart contract is $\mathbf{1.28}\boldsymbol{M\ gwei}$ if the gas price is $\mathbf{40}\ \boldsymbol{Wei}$, where the estimated gas consumed is $\mathbf{32000}\ \boldsymbol{units\ of\ gas}$. The research aims to determine the $(\boldsymbol{gc})$ for different types of transactions of different upgradeable patterns.

- $(\boldsymbol{gc})$ of Deployment $\boldsymbol{T_x}$

Since the deployment of an upgradeable contract creates more than two SCs, then the $(\boldsymbol{gc})$ depends on the number of transactions that created new contracts.
For example, in the case of implementing the **TransparentProxy** pattern, three SCs will be created according to Figure 1, and the gas consumption for three create transactions is $32{,}000 * 3$. Therefore, Equation (1) can be used to calculate the $(\boldsymbol{gc})$ of the creation transaction where (n) is present in the number of contracts created during the deployment process.
It is identified from Figure 8 that deploying the UUPSUpgradeable patterns is cheaper than the other patterns.

$$32{,}000 * n, n \in \mathbb{N}\ and\ n \geq 2 \qquad (1)$$

- $(\boldsymbol{gc})$ of Normal $\boldsymbol{T_x}$

The $CALL\ and\ DELEGATECALL$ are categorised as normal T_x because any IC functions are invoked through these transaction types. We assume the end-user has invoked the IC function, which will be delegated to IC through Px as a $DELEGATECALL$ transaction. The $(\boldsymbol{gc})$ of the entire process from the end-user to IC is calculated as four different transactions. According to Appendix G[3] gas consumption of every transaction costs $21{,}000$, then the total gas consumption is calculated as in Equation (2). The CALL and DELEGATE CALL have the same value of $(\boldsymbol{gc})$ because they are treated as a transaction.

$$gc_{call} + gc_{delegatecall} + \cdots \qquad (2)$$

All upgradeable patterns have the same number of transactions except UUPSUpgradeable as the proxy and IC combined as one smart contract. For that, it has less $(\boldsymbol{gc})$ compared with the remaining patterns.

- $(\boldsymbol{gc})$ of Upgrade $\boldsymbol{T_x}$

The upgrade T_x is a $\boldsymbol{CALL}$ transaction because the upgrade is the name of the function built within the $\boldsymbol{ProxyAdmin}$ contract. The EOA invokes this function through $\boldsymbol{ProxyAdmin}$ to upgrade the IC address. The $(\boldsymbol{gc})$ is determined by calculating the $\boldsymbol{CREATE}$ transaction of the new version with the $\boldsymbol{CALL}$ transaction of upgrading IC address. Figure 5 illustrates the number of transactions that can be called to execute the upgrade function.

$$gc_{call} + gc_{CREATE} \qquad (3)$$

The gas consumption for different contract transactions can vary depending on the transaction type and the contract code's complexity. For example, the deployment transactions consume a fixed amount of gas because this transaction only involves creating the contract and does not execute any of its functions. On the other hand, invoking a function in **IC** requires multiple transactions to be executed and complete the call. It includes an average transaction to initiate the call, internal transactions to execute the function, and a final transaction to return the result. Therefore, the gas consumption for these transactions can be more expensive than the deployment transaction.

Additionally, performing the upgrade to an IC, the owner must initiate a transaction to update the proxy contract by changing the address of the old version to the new version. This transaction consumes gas, as does the deployment transaction to deploy the new implementation contract. Therefore, the total gas consumption for an upgrade depends on the complexity of the new implementation contract and the number of transactions required to complete the upgrade.

[1] https://etherscan.io/gastracker.

4.2.2 Performance

In our research, we have conducted a qualitative analysis method to evaluate the performance of different patterns and perform the comparison between them. The evaluation estimates the number of transaction rounds required to execute the target function within the proxy patterns or IC. The comparison will be based on transaction types as it will cover in the following sections.

- **Performance of Deployment T_x**

The regular **deploymentT_x** of deploying contract requires one transaction, called creation Tx. In the case of deploying the different patterns, such as **ERC1967** pattern, it also required three rounds of transactions, as it initialised three different creation transactions. It is responsible for creating the **ERC1967Proxy**, **ERC1967Upgrade**, and **IC**. On the other hand, deploying the **Beacon** pattern required three rounds of creation transactions, as shown in Figure 6. Moreover, deploying the UUPSUpgradeable contract required two rounds of creation transactions as the UUPSUpgradeable is built-in with the IC and needs ERC1967Proxy to avoid the proxy selector clashing issue.

The performance of this type of transaction can be estimated by counting the number of rounds required to create each contract pattern successfully. However, each created contract is created in an individual block because once the IC is created, the address will be stored within the proxy pattern and used later as input to delegate the end-user invocation. For that, by using our analysis method, we assume that **UUPSUpgradeable** patterns have high performance as it requires only two rounds of creation transactions during the deployment process.

- **Performance of Normal T_x**

The **normal T_x** is the call initiated by the end-user to IC to execute the targeted function. In the case of executing a function of IC via proxy, it required six transactions. Those transactions represent different types of calls, starting with the call from the end-user to IC through the proxy. Then, once the IC receives it, it will be executed and delegated to the end-user. Finally, the proxy completes the final transaction to update the contract state once the process is done successfully. The internal transactions consist of three transactions which are linked with the original transaction. Once the output from internal transactions is delegated to the end-user, the six transactions will be added to the same block with the same hash value. Figure 5 shows the number of transactions performed to execute a function in the **ERC1967** pattern.

As the proxy plays an essential part in all patterns, we estimate that ERC1967 and Transparent Proxy patterns have the same number of transactions to execute the targeted function. Therefore, the UUPSUpgradeable pattern has the best performance as the execution of the function is done within the same contract.

Our method focuses on the number of transactions initiated to be executed. Other metrics can be considered experimentally to know which patterns perform better than others. We aim to evaluate the performance experimentally by measuring throughput, latency, and code complexity which can give us an accurate result.

- **Performance of Upgrade T_x**

Upgraded smart contracts involve the contract owner changing the address of IC from the old version to the new version. Figure 5 represents the upgrade process of the ERC1967 pattern, as the number of transactions performed is equal to the number of transactions performed by the Beacon pattern. We assume that both patterns perform similarly during the upgrade process.

In the case upgrade process done by the TransparentProxy, the owner calls the upgrade function similarly as the end-user invokes a function from IC. For that, we assumed that the number of transactions required for upgrade Tx is similar to normal Tx. Therefore, we assume that UUPSUpgradeable is best performed, followed by the TransparentProxy pattern, and the remaining patterns are performed slower.

4.2.3 Security

The design of an upgradeable contract does not prevent the known attack of smart contracts, such as reentrancy, because the main idea of an upgradeable contract is to ensure the immutability concept of blockchain by reserving the contract state.

In this section, the research aims to discover the security vulnerabilities that affect upgradeable patterns. First, we have used some datasets from Section 3. Then, we use the Slither [1] tool to analyse the source codes of different upgradeable patterns. Figure 9 shows the result of different affected vulnerabilities and their impacts on each pattern.

It shows that 20% of the detected vulnerabilities in **Beacon** and **TransparentProxy** patterns have an informational impact. However, the vulnerabilities categorised as informational will not affect the safety of upgradability because they do not affect the upgrade functions. On the other hand, the **ERC1967** pattern is affected by 60% of detected vulnerabilities equally divided between the informational, low, and medium impacts. For that, the code complexity of upgradeable patterns could be one of the main metrics that might **ERC1967** pattern to be the most affected, with 20% of vulnerabilities with medium impact.

There is no result related to **UUPSUpgradeable** patterns in Figure 9. For that, we need to use another analysis tool in future to verify the results.

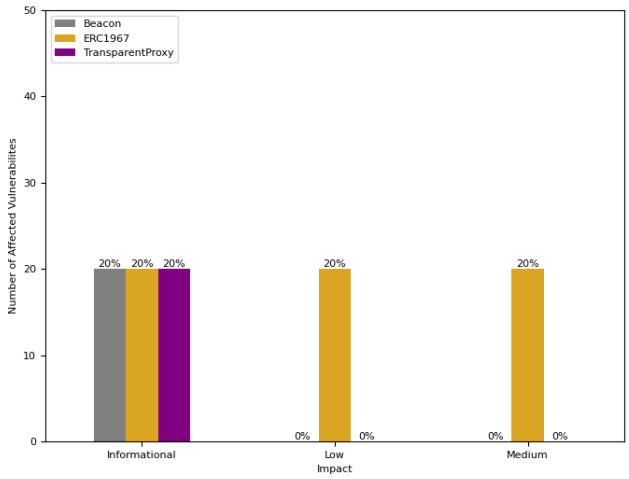

Figure 9. The number of affected vulnerabilities based on OpenZeppelin patterns

Therefore, the good practice to keep the contract safe and secure is developing the following mechanisms besides the upgradeable contract [9].

- **Ownable**
  This contract module provides an access control mechanism and ensures the authorisation of upgradeable contracts [9]. It is linked with upgradeable contracts, such as Transparent, Beacon, and UUPSUpgradeable contracts. The EOA who deploys this contract will be the owner by default unless he transfers the ownership to another owner by invoking the **transferOwnership( )** function, which is only reachable by the owner. If the ProxyAdmin account needs to be changed, it will be done with the Ownable contract's function.
- **ReentrancyGuard**
  This contract module is designed to prevent reentrant calls to the contract function [9]. It is built with the **nonReentrant( )** modifier, which fails any call execution with a reentrancy pattern.

5 Analyse the Usage of OpenZeppelin Upgradeable Smart Contracts Over the Last Four Years

In this section, we analyse the use of OpenZeppelin Upgradeable smart contracts over the last four years. The analysis starts by using the dataset which been created from Section 3.3.

5.1 Result Analysis

Figure 10 compares the number of upgradeable patterns and how they evolved over the last four years. For example, the **TransparentProxy** pattern's value increased significantly from 2019 to 2022, with a value of 5116 in 2019 and a value of 53,898 in 2022. These changes represent a growth of more than 90%. Similarly, the value of the **UUPSUpgradeable** pattern increased from 122 in 2019 to 13,134 in 2022.

On the other hand, the value of the **ERC1967** pattern remained relatively constant from 2019 to 2021 but increased significantly in 2022. However, the value of the **Beacon** pattern also increased steadily from 2019 to 2022, with a value of 9 in 2019 to 229 in 2022.

Overall, the data show that the Transparent and UUPSUpgradeable patterns are the most upgradeable and are used between 2019 and 2022. The comparisons discussed earlier in this paper could be related. It was clear that Transparent patterns have been invented to overcome the issue discussed in Section 2.2. In addition, the UUPSUpgradeable pattern has better performance and is cheaper. The values of the ERC1967 and Beacon patterns have remained relatively constant or have increased steadily over the four years. This analysis provides empirical results of using upgradeable smart contract patterns based on the OpenZeppelin technique.

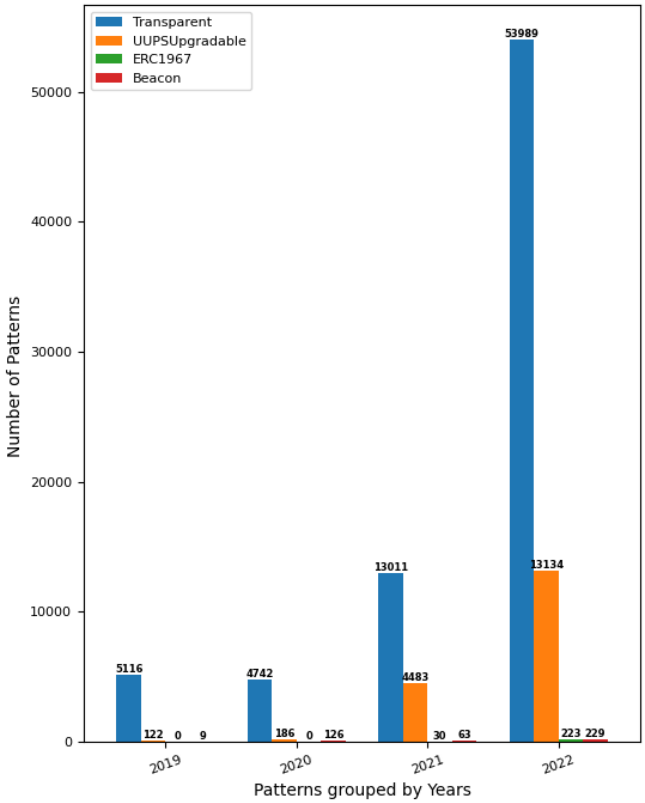

Figure 10 Number of Upgradeable patterns between 2019 and 2022

6 Conclusion and Future Work

In this paper, we reviewed the OpenZeppelin Upgradeable patterns and compared them from different aspects, such as cost, performance, and security. We performed some analyses to find the vulnerabilities that might affect upgradeable smart contract patterns.

The difference between upgradeable patterns on performance and gas consumption is performed based on three

transactions: DeploymentTx, NormalTx, and UpgradeTx. However, the comparison between upgradeable patterns on performance was based on a qualitative analysis. We found that the UUPSUpgradeable pattern performs better according to the number of transactions completed per round. In the case of gas consumption, the gas consumption while using the UUPSUpgradeable pattern is cheaper than other patterns. We assume that the Beacon pattern also consumes much gas and has slow performance according to the number of transactions performed during the interaction. Finally, the result analysis shows that the use of Transparent Proxy of upgradeable patterns has grown significantly over the last four years.

In future work, we aim to verify the vulnerabilities identified in Section 4.2.3 by creating a threat model and identifying upgradeable patterns' security requirements. In addition, we will conduct experimental evaluations of the performance of the upgradeable patterns. The experimental method will involve measuring key metrics such as throughput, latency, and code complexity and using these metrics to compare the performance of different upgradeable patterns.

Competing Interests:
None declared.

Ethical approval:
Not applicable.

Author's contribution:
The authors worked together to design and conduct this research and prepare the manuscript.

Funding:
University of Southampton
Ministry of Higher Education, Research Innovation Oman

Acknowledgements:
Not applicable.

References

[1] 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, May 2019. pp. 8-15.

[2] Ethereum, "EIPS/EIP-1967.MD at master Ethereum/EIPS," GitHub, 08 Sep. 2022. [Online]. Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1967.md.

[3] "Yellow paper – github pages." [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf.

[4] G. BigQuery, "Ethereum blockchain," Kaggle, 04 Mar. 2019. [Online]. Available: https://www.kaggle.com/datasets/bigquery/ethereum-blockchain.

[5] W. Jie, A. S. Koe, P. Huang, and S. Zhang, "Full-stack hierarchical fusion of static features for smart contracts vulnerability detection," *2021 IEEE International Conference on Blockchain (Blockchain)*, 2021. Availbale: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9680540.

[6] "Proxies," *OpenZeppelin Docs*. [Online]. Available: https://docs.openzeppelin.com/contracts/4.x/api/proxy.

[7] "Proxy upgrade pattern," *OpenZeppelin Docs*. [Online]. Available: https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies?utm_source=zos&utm_medium=blog&utm_campaign=proxy-pattern#the-constructor-caveat.

[8] S. Palladino, "The transparent proxy pattern," *OpenZeppelin blog*, 26 Apr. 2021. [Online]. Available: https://blog.openzeppelin.com/the-transparent-proxy-pattern/.

[9] "Security," *OpenZeppelin Docs*. [Online]. Available: https://docs.openzeppelin.com/contracts/4.x/api/security.

[10] M. Wohrer and U. Zdun, "Smart contracts: Security patterns in the ethereum ecosystem and solidity," in *2018 IEEE 1st Int. Work. Blockchain Oriented Softw. Eng. IWBOSE 2018 - Proc.*, vol. 2018-Janua. IEEE, 2018. pp. 2–8.

[11] "Proxies," *OpenZeppelin Docs*. [Online]. Available: https://docs.openzeppelin.com/contracts/4.x/api/proxy.

[12] S. Palladino, Aquiladev, and Josselinfeist, "Contract upgrade anti-patterns," *Trail of Bits Blog*, 06 Sep. 2018. [Online]. Available: https://blog.trailofbits.com/2018/09/05/contract-upgrade-anti-patterns/.

