Not so immutable: Upgradeability of Smart Contacts on Ethereum

Mehdi Salehi[1], Jeremy Clark[1], and Mohammad Mannan[1]

Concordia University, Montreal, Canada

Abstract. A smart contract that is deployed to a blockchain system like Ethereum is, under reasonable circumstances, expected to be immutable and tamper-proof. This is both a feature (promoting integrity and transparency) and a bug (preventing security patches and feature updates). Modern smart contracts use software tricks to enable upgradability, raising the question of *how* is upgradeability achieved and *who* is authorized to change the contract. In this paper, we evaluate seven upgradability patterns for security, usability, and deployability. We develop a measurement framework for finding all upgradeable contracts of the most prominent type and examine how they implement access control over their upgradability: about 50% are are controlled by a single Externally Owned Address (EOA), and about 20% are controlled by multi-signature wallets in which a limited number of persons can change the whole logic of the contract.

1 Introductory Remarks

Capitalization, infrastructure, and regulation make new deployments of financial technology challenging. Blockchain systems like Ethereum radically reduce the barrier-to-entry by allowing anyone to deploy any solution, provided it can operate autonomously as software. The key promise of code running on Ethereum is that the code will execute exactly as it is written, and the code that is written can never be changed. Ethereum cannot maintain this promise unconditionally but its assumptions—that certain cryptographic primitives are secure, and that well-intentioned participants outweigh malicious ones—provide a realistic level of assurance.

The immutability of a smart contract's code is related to trust. If Alice can validate the code of a contract, she can trust her money to it and not be surprised by its behaviour. Unfortunately, disguising malicious behaviour in innocuous-looking code is possible, and many blockchain users have been victims of such contracts ('rug pulls'). On the other hand, if the smart contract is long-standing with lots of attention, and perhaps assessments from third-party professional auditors, the immutability of the code can add confidence.

Consider the case where a security vulnerability in the code of a smart contract is discovered. The flip-side of immutability is that it prevents software updates. Less urgently, some software projects may want to roll out new features, which is also blocked by immutability. There is an intense debate about

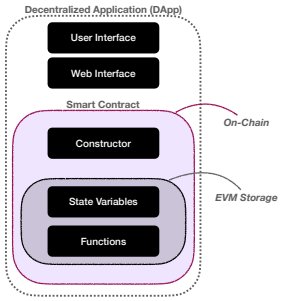

Fig. 1. Components of a decentralized application.

whether this is a positive or negative, with many claiming that 'upgradability is a bug.' We do not take a position on this debate. We note that upgradability is happening and we seek to study what is already being done and what is possible.

Is there a way to deploy upgradeable smart contracts if all smart contracts are (practically speaking) immutable? Consider a two simple ideas. The first is to deploy the upgraded smart contract at a new address. One main drawback to this is that all software and websites need to update their addresses. A second simple idea is to use a proxy contract (call it P) that stores the address of the 'real' contract (call it A). Users consider the system to deployed at P (and might not even be aware it is proxy). When a function is called on P, it is forwarded to A. When an upgrade is deployed to a new address (call it B), the address in P is changed from A to B. This solution also has drawbacks. For example, if the proxy contract hardcodes the list of functions that might be called on A, new functions cannot be added to B. Another issues is that the data (contract state) is stored in A. For most applications, a snapshot of A's state will need to be copied to B without creating race conditions. Mitigating these issues leads to more elaborate solutions like splitting contract logic and state between different contracts, utilizing Ethereum-specific tricks (fallback functions to capture unexpected function names), and trying to reduce the inefficiencies of using a cluster of contracts.

Contributions. Finalize this text later.

2 Background and Related Work

Updating vs. upgrading. Software maintenance is part of software's lifecycle, and the process of changing the product after delivery. Often a distinction is drawn between software *updates* and software *upgrades*. An update modifies isolated portions of the software to fix bugs and vulnerabilities. An upgrade is generally a larger overhaul of the software with significant changes to features and capabilities. In our paper, we will only use the term upgrade and instead distinguish between retail (parameters and isolated code) and wholesale (entire application) changes.

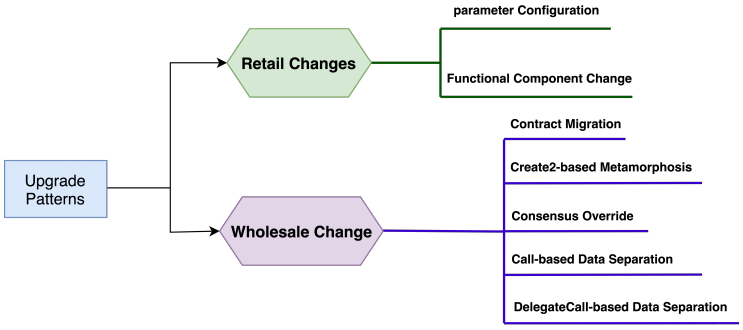

Fig. 2. Classification. Re-arrange to match ordering of text. Include section numbers of leaf nodes.

DApp vs. smart contract. Figure 1 shows the main components of a decentralized application (DApp). The core component is the smart contract (or simply contract), which is the set of functions and state stored on-chain. When first deployed, the smart contract also includes a constructor function which executes once and is then discarded (to be more precise, a copy of the constructor is stored in the record of transactions, called the calldata, but it is not retained in the EVM and can never be called again). While it is possible to interact directly with a smart contract by invoking its functions through Ethereum, generally users are provided an off-chain website with a user interface. Website actions are translated into calls to the Ethereum network through a set of tools (most prominently web3) in the web interface.

While upgrades to the user interface can significantly change a user experience and expose new features, they are governed by traditional software maintenance. Our paper only considers the on-chain smart contract component, which is significantly more challenging to upgrade as it is on-chain and immutable under reasonable circumstances.

Related work. TBD

3 Classification of Upgrade Patterns

A variety of upgradeability patterns have been proposed for smart contracts on Ethereum. We categorize them in Figure 2. While some distinctions we make are applicable to other blockchain systems or even to software in general, the most popular approaches are leveraging Ethereum-specific operations and memory layouts.

The approaches have evolved over time and some of them are no longer useful because of the advantages of other new methods. Can we prove this with measurements?

3.1 Parameter Configuration

We first categorized upgradeability patterns into two main classes: *retail changes* and *wholesale changes*. A pattern for retail change does not enable the replacement of the entire contract. Rather, a component of the contract is predetermined (before the contract is deployed on Ethereum) to allow future upgrades, and the code is adjusted to allow these changes.

The simplest upgrade pattern is to allow a system parameter, that is stored in a state variable, to be changed. This requires a *setter function* to overwrite (or otherwise adjust) the variable, and access control over who can invoke the function. For example, in decentralized finance (DeFi), many services have parameters that control fees, interest rates, liquidation levels, *etc.*. Adjustments to these parameters can initiate large changes in how the service is used (its 'tokenomics'). A DeFi provider can retain control over these parameters, democratize control to a set of token holders (*e.g.,* stability fees in the stablecoin project MakerDao), or lock the parameters from anyone's control. In Section **??**, we dive deeper into the question who can upgrade a contract.

3.2 Functional Component Change

While a parameter change allows an authorized user to overwrite memory, a functional component change addresses modifications to the code of a function (and thus, the logic of the contract). In the EVM, code cannot be modified once written and so new code must be deployed to a new contract, but can be arranged to be called from the original contract.

One way to allow upgradable functions is deploying a helper contract that contains the code for the functions to be upgradeable. Users are given the address of the primary contract, and the address of this secondary (helper) contract is stored as a variable in the primary contract. Whenever thhis function is invoked at the primary contract, the primary contract is pre-programmed to forward the function call, using the opcode `Call`, to the address it has stored for the secondary contract. To modify the logic of the function, a new secondary contract is deployed at a new address, and an authorized set of individuals can then use a parameter change in the primary contract to update the address of the secondary contract.

The DeFi lending platform Compound uses this pattern for their interest rate models which are tailored specifically for each asset. The model for one asset can be changed without impacting the rest of the contract.

Upgradeable functional components need to be pre-determined before deploying the primary contract. Once the primary contract is deployed, it is not possible to add upgradeability to existing (non-upgradable) functions. It also cannot be directly used to add new functions to a contract. Finally, this pattern is most straightforward when the primary contract only uses the return value from the function to modify its own state. Thus, the function is either 'pure' (relies only on the parameters to determine the output) or 'view' (can read state from itself or other contracts, but cannot write state). If the function modifies

the state of the primary contract, the primary contract must either expose its state variables to the secondary contract (by implementing setter functions), or it can run the function using `DelgateCall` if the secondary contract has no state of its own.

This upgrade pattern suggests a way forward for wholesale changes to the entire contract: create a generic 'proxy' contract that forwards all functions to a secondary contract. To work seamlessly, this requires some further engineering (see sections **??** and **??**)) but will see such proxy contracts based on `Call` and `DelegateCall` shortly.

3.3 Contract Migration

The two previous patterns enable portions of a smart contract to be modified. The remaining patterns strive to allow an entire contract to be modified or, more simply, replaced. The simplest pattern is to deploy a new version of the contract at a new address, and then inform users to use the new version—sometimes called a 'social upgrade.' One example is Uniswap, which is on version 3 at the time of writing. Versions 1 and 2 are still operable at their original addresses.

Contract migration does not require developers to instrument their contracts with complex logic, as in the data separation patterns below, which can ease auditability and gas costs for using the contract. However, depending on the application, there may be a need to transfer the data stored in the old contract to the new version. This is generally done in one of two ways. The first is to collate the state of the old contract off-chain and place it into the new contract's constructor (or split it into multiple function calls, as needed). If the old contract was instrumented with an ability to pause it, this can eliminate race-conditions that could otherwise be problematic during the data migration phase. The second method is specific for certain types of data like a user's balance of tokens. For things like this, a manual migration can be initiated by (and paid in gas fees by) the user when they want to move their balance from the old contract to the new contract.

3.4 Create2-based Metamorphosis

Is it possible to do contract migration, but deploy the new contract to the *same* address as the original contract, effectively overwriting it? If so, developers can dispense with the need for a social upgrade (but would still need to accomplish data migration). At first glance, this should not be possible on Ethereum, however a set of opcodes can be 'abused' to allow it: specifically, the controversial[1] `SELFDESTRUCT` opcode and the 2019-deployed `CREATE2`.

Consider a contract, called Factory, that has the bytecode of another contract, A, that Factory wants to deploy at A's own address. `CREATE2`, which supplements the original opcode `CREATE`, provides the ability for Factory to do

[1] "Expectations for backwards-incompatible changes / removal of features that may come soon." V. Buterin, Reddit r/ethereum, Mar 2021.

this and know in advance what address will be assigned to contract A, invariant to when and how many other contracts that Factory might deploy. The address is a structured hash of A's "initialization" bytecode, parameters passed to this code, the factory contract's address, and a salt value chosen by the factory contract.[2] Most often, A's initialization bytecode contains a copy of A's actual code ("runtime" bytecode) to be stored on the EVM, and the initialization code is prepended with a simple routine to copy the runtime code from the transaction data (calldata) into memory and return. Importantly, however, the initialization bytecode might not contain A's runtime bytecode at all, as long as it is able to fetch a copy of it from some location on the blockchain and load it into memory. In order for `CREATE2` to complete, the address must be empty, which means either (1) no contract has ever been deployed there, or (2) a contract was deployed but invoked `SELFDESTRUCT`.

Assume the developer wants to deploy contract A using metamorphosis and later update it to contract B.[3] The developer first deploys a factory contract with a function that accepts A's (runtime) bytecode as a parameter (which includes the ability to self destruct). The factory then deploys A at an arbitrary address and stores the address in a variable called codeLocation. The factory then deploys a simple 'transient' contract using `CREATE2` at address T. This contract performs a callback to the factory contract, asks for factory.codeLocation, and copies the code it finds there into its own storage for its runtime bytecode and returns. As a consequence, A's bytecode is now deployed at address T.

To upgrade to contract B, the developer calls `SELFDESTRUCT` on A. It then calls the factory with contract B's bytecode. The factory executes the same way placing a pointer to B in factory.codeLocation. Importantly, it generates the same address T when it invokes CREATE2 since the 'transient' contract is identical to what it was the first time—this contract does not contain contract A or B's runtime code, it just contains abstract instructions on how to load code. The result is contract B's runtime bytecode being deployed at address T where contact A was.

As it is concerning that a contract's code could completely change, we note that metamorphic upgrades can be ruled out for any contract where either: it was not created with `CREATE2`, it does not implement `SELFDESTRUCT`, and/or its constructor is not able to dynamically modify its runtime bytecode.

3.5 Consensus Override

This wholesale pattern is not a tenable solution to upgradeability as it as only been used rarely under extraordinary circumstances, but we include it for completeness. Immutability is enforced by the consensus of the blockchain network. If participating nodes (*e.g.,* miners) agreed to suspend immutability, they can in theory allow changes to a contract's logic and/or state. If agreement is not

[2] Specifically: $\mathsf{addr} \leftarrow \mathcal{H}(\texttt{0xff} \| \mathsf{factoryAddr} \| \mathsf{salt} \| \mathcal{H}(\mathsf{initBytecode} \| \mathsf{initBytecodeParams}))$

[3] Medium: `https://medium.com/@0age/the-promise-and-the-peril-of-metamorphic-contracts-9eb8b8413c5e`

unanimous, the blockchain can be forked into two systems—one with the change and one without.

In 2016, a significant security breach of a decentralized application called 'the DAO' caused the Ethereum project to propose overriding the immutability of this particular smart contract to reverse the impacts of attack. In the unusual circumstances of this case, it was possible to propose and deploy the fix before the stolen ETH could be extracted from the contract and circulated. Nodes with a philosophical objection to overriding immutability continued operating, without deploying the fix, under the name Ethereum Classic.

3.6 Call-based Data Separation

Generally, a developer needs to predetermine that the contract will be upgradeable in this wholesale fashion before deploying it, but once the upgradeability pattern is set, any aspect of the contract's logic can be changed. This enables security fixes and new features, but could also lead to fraud and 'rug pulls.'

Patterns The other type of wholesale methods is to separate data and logic part of our codes so, in the upgrade event we can keep the storage contract and just deploy a new logic contract and link the storage to the new version of logic contract. There is a debate on whether this type of upgradeability is cheaper or not in comparison to migration method. But, this method is more efficient for Dapps in which we need frequent updates. The other important issue here is who decides on the changes we need for the system which we will discuss on further sections. Here we have 2 different choices using *Call* method and *Delegate Call* method to link storage and logic contracts together. We will dive deeper into these two approaches:

This type is also known as data separation pattern. In this type the interaction between logic and storage contact is handled by Call opcode in Ethereum Virtual Machine (EVM). In call based patterns user is supposed to call the logic contract and the logic contract will interact with the storage by calling setter or getter function inside storage contract whenever needs to read or write a data. The logic contract is the one that should be upgraded.

The most important part in this pattern is that we cannot change the storage layout of the storage contract after deployment. So we cannot add new storage variable after the deployment. But, in most smart contracts we need to add new data for example adding the balance for new user. A solution to this problem is proposed in ERC930 or Eternal Storage pattern. Eternal storage uses mapping (key-value pair) to store data, using one mapping per data type. For instance for integer type we will define a mapping of byte32 to integer. This way we always can add new integer variables to our storage contract just by adding a variable to this mapping and having a new byte32 as key. This pattern is a good solution when we have simple data types like integer, boolean, string, etc. but it is very complex to have a eternal storage layout for complex data types such as structures, mappings or combination of them.

In this type of upgrade the address of the contract will be changed after upgrade so we need to aware our users about the change and interacting with the new version. Also it may break the compatibility of the ecosystem. In case of upgrade all smart contracts and Dapps that are interacting with the upgraded smart contract must change the address which they pointed to in order to interact with our contract. It may leads into a disaster if other contracts that are interacting with our contract do not have a way to change the address. We should also make other off-chain services (e.g. exchanges) aware of the change to start using the new version of your contract. In Call-based pattern we should have a way to stop previous version during/after upgrade because both of them are shared the same storage contract. There are three main ways to implement upgrades using data separation pattern. The easiest way is to change the ownership of storage contract into new upgraded logic contract and then *Pause* the old contract using *Circuit Breaker* pattern or set its pointer to 0x0 address. The other solution is to forward the calls receive by the old contract into the new logic contract. The last option is to set a registry contract that just keeps the address of latest version of the logic contract and call into it.

Using Calls-based pattern we eliminate the process of data migration from the old contract to the newer version and it is easy to understand this type of upgradeability pattern. But it is hard for developers to deal with this pattern when their logic contract needs complex data structures such as mapping or structures. Also the developers should change their code a lot if they decide to use this upgradeability pattern in their non-upgradeable code.

3.7 DelegateCall-based Data Separation

Similar to call based patterns here we have two contracts, Storage and Logic contract. In this pattern we may have more than one logic contract or storage contract. The difference here is that the user is calling storage contract(s) first (a.k.a proxy contract), and the proxy contract will use *DelegateCall* opcode to link to the logic contract(s) (a.k.a implementation contract). In fact, there is a fallback function inside the proxy contract and inside the proxy, there is a delegate call that forwards the whole message data to the implementation contract. The fallback functions in Ethereum will be executed if the user calls the smart contract with a function signature that does not exists on that contract.

So, if the user calls a function that does not exists on the proxy contract, the fallback will be executed and so the message data will be delegated call into the implementation contract and if the function that is called by user exists on the implementation contract the function will be executed.

So if we need to add/change the functionality of our Dapp, we just need to deploy a new implementation contract and then change the implementation address inside the proxy contract to delegate call to the new version of the implementation.

Using this pattern has two major limitations that we should take care of:

First, as described above the pattern has an assumption that the function signature of the functions inside the implementation is not existed inside the

proxy contract as well. Otherwise when a user calls that function the fallback function won't be executed because the function exists on the proxy contract. So that function is not achievable inside the proxy. In Ethereum when we want to call a function we just use 4 Bytes of the hash of function name and input/output types as the function signatures. So it is possible to have 2 completely different functions with the same function signature. In the case that we have a function in implementation contract that has the same function signature with one of the functions inside the proxy we say that function signature clashes happened.

Second, the delegatecall opcode in Ethereum will run the logic of the implementation contract in the context of the proxy contract. It is similar to copy pasting the logic code into the proxy contract and altering the storage of the proxy contract. Because the delegate call opcode preserves the context, we should be sure that the storage layout of the proxy contract and the logic contract should be the same. Otherwise when we try to change a storage slot in proxy, we will end up with changing another one. The difference between storage layouts will result in a storage clashes.

Different proxy patterns are proposed to mitigate function selector clashes and storage clashes. We will describe each of them and the advantage and disadvantages of each of them.

The first problem that we should deal with in proxy contracts is function selector clashes. In fact inside the proxy contract we need just one function called *upgradeTo* function that gives the owner of the contract ability to change the address of the implementation contract inside the proxy. So in the event of the upgrade, the owner will deploy the new implementation contract and change the address variable inside the proxy to the address of the newly deployed implementation contract. By doing this the fallback function will delegate call into the newer version. So to mitigate the problem of function signature clashing we should be sure that we do not have a function with function signature equal to the functions inside the proxy, we have two main approaches to mitigate this problem:

Universal Upgradeable Proxy Standard (UUPS) . The Universal Upgradeable Proxy Standard (UUPS) method is proposed on 2019-03-04 and suggested to move the upgradeTo function to the implementation contract. So by doing this we do not have any function inside the proxy and the function signature will be mitigated. Also we reduced the size of the proxy contract. If the owner decides to upgrade the system, should call the upgrade function via calling the proxy and because the code is executed in the context of the proxy contract, the address variable of implementation inside the proxy contract will be changed. There is a huge risk in using the UUPS proxy pattern. If the owner upgrades to a contract that does not implemented an upgradeTo functionality inside it, then the contract cannot be upgraded after that last upgrade because there is not any function inside the proxy and the latest implementation contract to upgrade the system. There is a way to check if the proposed address is a contract address and also the upgradeTo function is implemented inside that or not but that does not guarantee that the logic of the upgradeTo function is what needed to be.

Transparent Proxy . The other way to mitigate the function clashes is to check the sender of the transaction. If the owner of the contract calls the proxy the upgrade function will be executed and otherwise the fallback will be executed. This way we are sure that even if we have function clashes, then user's call will be forwarded to the implementation. The drawback of using Transparent proxy is that we add a check for each call to the contract which added gas cost for users on each call to the contract. (it needs to first read the address of the owner from storage which needs high amount of gas and then a check that if the sender is owner or not).

There are three different methods to mitigate the risk of storage clashes. To mitigate storage clashes we have three different approaches :

Inherited Storage . In this method the proxy and all logic contracts are inherited from a storage contract that contains storage variables. If we decided to upgrade the contract, we should be sure that the new implementation contract is inherited from the storage contract. Using this method we are confident that the proxy and logic contracts are using the same storage layout and storage clashes will be mitigated. Also if after deployment we need to add new storage variables, we should just deploy a new storage contract that inherited from the previous storage contract and add the new variables to it. We should be sure that the future implementation contracts will inherit the latest storage contract. This adds-on to the inherited storage contract is called append-only pattern. This method is not efficient because of variables that declared but not used in some logic contracts. On the other hand, each logic contract is coupled with a storage contract and it is hard to take care of this track. Also we should take care of upgrading the system each time to be sure that the new implementation contract is inherited from the latest version of storage contract.

Eternal Storage . As described before in Eternal storage, we defined mappings for all variable types that we need to use in our logic smart contract. For storing mapping variables EVM selects random slots on the storage based on the variable's name so we can mitigate the clashes using this randomness. The main problem of this type is that the logic contract and all other contracts that are using the storage must use the mapping structure to access the storage variables and use complex syntax whenever they want to access a variable. This also results in the gas usage inefficiency because we need to call and update a mapping each time we need to change a variable.

Unstructured Storage . The other way of mitigating the storage clashes is to assign some randomly selected slots to critical variables like address of logic contract. For instance, openzeppelin uses hash of "org.zeppelinos.proxy.implementation" to store the address of the logic contract in this slot. The downside of this approach is that we need getter and setter function for each variable. We also can use unstructured storage for simple variables and not for mapping and structures. EIP-1967 proposed to assign specific storage slots for address variable inside the proxy contract to store the address of the implementation contract inside the

proxy. The proposed slot is 0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc which is calculated from this equation bytes32(uint256(keccak256(eip1967.proxy.implementation)) - 1)).

There are two other upgradeable proxy patterns that are proposed to address problems in some specific applications. We will describe them bellow:

Beacon Proxy . In some type of applications such as wallets, the logics needs to make the data of each user separate. There are some proposals such as EIP-1167 or minimal proxy suggested creating a proxy contract for each user that delegate calls to the main implementation contract. Using this approach each user has its own data inside its proxy contract. The problem of using the minimal proxies is that they are not upgradeable. If a bug found in the implementation contract then all proxy instances are prone to the attack. The problem of using the previous types of the upgradeability patterns is that unlike the previous ones that have just one proxy contract and easy to change the address variable inside them, here we have tons of instances and we cannot change all address variables inside all proxies.

Beacon proxy method is suggested to solve this issue. In beacon proxy, each proxy instance will call a registry-like contract and ask it for the latest version of the implementation contract and then will delegatecall to the resolved address. In the upgrade event, the owner needs to just change the address inside the beacon contract so all proxy instances will use the newer version of implementation contract each time calling the beacon contract and asking for the implementation contract address.

Diamonds The EIP-2535 or Diamonds proxy is proposed on 2020-02-22 and suggested using multi implementation contracts with a single proxy contract. In the proxy there is a access control structure in which there is a mapping between each implementation contract's address and the function signatures that are implemented inside that specific implementation contract. Using this method we can have a separate implementation contract for each functionality of the Dapp. It will help to make the contracts more modularize. Also we can just update one functionality in each upgrade event. It also helps with the situation that the contract code size exceeds the limitation (24KB) by splitting it into a number of implementation contracts.

The drawback of using Diamonds is adding more complexity to the system because using different implementation contracts will increase the chance of storage clashes and error in handling the shared storage between them.

3.8 Evaluation Framework

Table **??** summarizes the pros and cons of each upgradability pattern. The details of the evaluation are provided in the full version of this paper.[4]

[1] Design of system in which a parameter can change the logic is hard

[4] To be archived. Can be provided anonymously though program chairs.

	Can replace entire logic	can replace pre-specified part of logic	Can replace entire state	can replace pre-specified state variables	No need to deploy a new contract	No need to migrate state from old contract	No need to separate State and Logic	Function Selector Clashes Risk	Storage Clashes Risk	No indirection	User endpoint address not changed	Downtime in upgrade events	No need to change code to add the upgrade pattern	Need to change a state variable
Parameter Configuration				✓	✓	✓	✓			✓	✓		✓	✓
Component Change		✓				✓	✓			⊠	✓		⊠	✓
Contract Migration	✓		✓				✓			✓			✓	
Create2 metamorphosis	✓		✓				✓			✓	✓	✓	✓	
Consensus Override	✓		✓				✓			✓	✓	✓	✓	
Call-based	✓					✓								✓
DelegateCall-based	✓					✓		✓	✓		✓		⊠	✓

Table 1. Evaluation**??**

4 Finding Upgradeable Contracts on Ethereum

This section aims to shed light on the state of upgradeable smart contracts on the Ethereum blockchain. Between all the different patterns described in previous sections, we focused on finding the *Delegate-call* based upgradeable contracts because it is the most widely used pattern for smart contract upgradeability at the time of writing this paper. The number of Ethereum Improvement Proposals (EIPs) and standards proposed for standardizing this pattern (e.g., EIP-1967, EIP-1822, EIP-2535, etc.) confirms this point. In the further parts of this section, we focus on describing the methodology used for finding upgradeable contracts in Ethereum blockchain, which use *Delegatecall-based* patterns and show the results. I have commented the detailed reason for choosing DelegateCall based pattern but removed it from the paper to summarize it

4.1 Methodology

As mentioned in classification section, the *DelegateCall-based* upgradeability approach consists of a storage contract (a.k.a proxy contract) and a logic contract (a.k.a implementation contract). The proxy contract is a simple type of smart contract in which there is a *Fallback* function. Fallback is a function inside smart contracts that do not have a function name. If a user sends a transaction to a contract to call a function that does not exist, it will pass into the fallback function, and the logic inside the fallback function will be executed. So, calling a

contract with a function signature that does not exist on that contract is equal to calling the contract's fallback function. Inside the fallback function of a proxy contract, there is a delegate call to the address of *implementation contract* (we call it *Target address* in the rest of the paper) which passes the whole data of the transaction to the implementation contract without altering it.

All proxy contracts have the above structure. However, *Upgradeable proxy contracts* should have another extra condition as well. The agent who is responsible for changes in the smart contract (a.k.a *admin*) must have the ability to change the target address. If a proxy does not have this condition, the contract delegates the data into a fixed implementation contract for the rest of its life. So, this type of proxies is not upgradeable. There are a bunch of patterns that follow this structure (e.g., Minimal Proxies [6], Delegate call forwarders [3], etc.), which we call them *Forwarders* in the rest of the paper. So, for upgradeable proxies, the target address must be *changeable*.

To find the proxy contracts in Ethereum, we need to collect transactions and all information regarding those transactions. To collect the transaction details, we need to replay the transaction and collect the data of execution of the transaction. Ethereum full archival node has a method, *trace_transaction*, that gives the traces [5] of the transactions executed on the specific block. We used this method to have transaction traces and find the transactions in which a delegate call happened. Each transaction trace may consist of several sub-traces (a.k.a, actions). If the data of two consecutive sub-traces of a transaction are equal and a delegate call is in the second sub-trace, it shows that the transaction passes the fallback function. Because if any other function in the contract is called (other than fallback), then the first four bytes of the data will be changed. Also, a delegate call in the fallback transferred the whole data without altering it, which means the contract is a proxy contract.

As discussed above, these proxies can be forwarders or upgradeable proxies. To find upgradeable proxies, we should filter them by checking whether the target address variable is changeable or not. Three general standards are proposed to change the target address of a proxy; Beacon proxy, Regular proxy, and Universal Upgradeable proxy. As discussed in the classification part, the target address in beacon proxies comes from an external call to another contract named *Beacon Contract*. So, to find upgradeable beacon contracts, we first check if the target address comes from an external call to another contract. If yes, we should check the callee contract to find out if the target address inside the beacon contract is changeable. If the target address is changeable, the proxy contract is a *Beacon Proxy* contract, and the admin of the beacon contract can change the target address inside it to upgrade the system.

If the address does not come from an external call, we check if there is any function inside the proxy contract that the admin can call to change the target address and upgrade the system. This is the most tricky part in our methodology to find out if a function inside the contract gives the admin the ability to change the target address. If that function is found, we mark the proxy

[5] Parity VM transaction trace

as an upgradeable proxy contract. The process is divided into two main parts; 1) Finding the target storage variable regarding the target address, and 2) checking if there is an assignment to that specific storage variable inside the contract.

Finding storage variable (slot) of the target address. We use bytecode decompiler named *Panoramix decompiler* [6] to decompile the bytecode of the contract into well-formatted python language codes. Then check to find the line of the code in which the delegate call happened and pick the target of the delegate call. We find the variable name or a storage slot of the target address, which is our goal in this part.

Checking for assignment. Now that we have the decompiled code and variable name (or storage slot) of the target address. We use the Regular Expression method to check if an assignment to that variable/slot happened in any function in the contract. If any assignment is found, we should be sure that the other variable assigned to the target address variable comes from the input of that function. If these conditions are satisfied, there is a function inside the contract that can change the target address and upgrade our system (upgrade function).

So by applying the first filter, we find the storage variable/slot of the target address and then check if it is changeable or not. If the target address is changeable, we mark the proxy as an upgradeable proxy contract. If there is no way to change the target address inside the proxy, we pass it to another final filter. There is another way of implementing upgradeable contracts named *Universal Upgradeable Proxy Standard (UUPS)* that is discussed in the classification section. In this method, the target address is changeable using the implementation contract. So to filter and find them, we check the implementation contract to find out if there is any function inside the implementation contract by which the admin can change the target address. If yes, then our proxy is a UUPS proxy contract. Otherwise, the proxy is not upgradeable. The first step here is to find the storage slot of the target address inside the proxy contract. Then we decompile the bytecode of the implementation contract and check to find if any assignment to that storage slot happened inside the implementation contract. The process of finding the assignment is very similar to the last part. So if a function is found that gives the admin a chance to write a new amount to the storage slot regarding the target address, the admin can call that function using the proxy to change the target address and upgrade the system. In this case, we marked the proxy as a UUPS proxy contract. All the remained proxy contracts are marked as non-upgradeable proxy contracts. The whole process is depicted in figure 4.1. For a detailed explanation of the methodology and implementation, check the appendix.

Results Having access to an Ethereum full archival node, we have collected transaction traces of transactions included in 2,064,595 blocks of Ethereum blockchain, starting from block number *#10800000* to *#12864595*. It covers transactions on the Ethereum blockchain from *Sep-05-2020* to *Jul-20-2021*.

[6] `https://github.com/palkeo/panoramix`

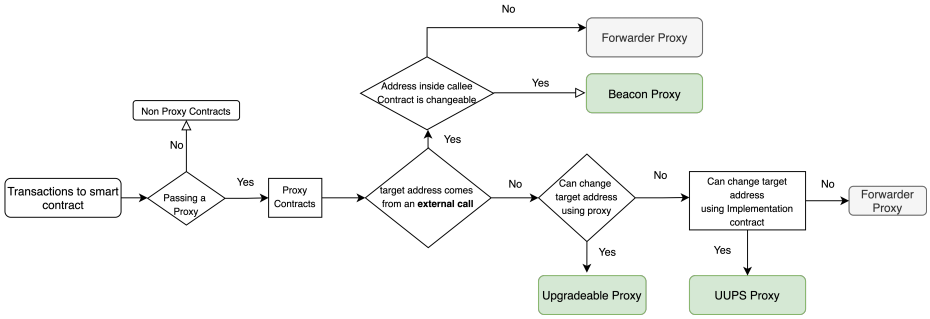

Fig. 3. Flowchart of the Process

Applying our methodology gives us *1,427,215* unique proxy contracts. However, a bunch of these proxies is using shared implementation contracts. We decide to weed out the proxy contracts that share the same implementation contract; however, two different Dapps may use the same implementation contract. The reason for this decision is that there are some Dapps like opensea [7] that create proxy contracts for each of their user, and these proxies share the same implementation contract (it will reduce the redundancy). So after filtering proxies with the same implementation contracts, we come up with *13,088* contracts. Afterwards we filter *Forwarder Contracts* from our dataset and totally we find *7,470* regular upgradeable proxy contracts. On the other hand, checking the remained proxy contracts and applying the methodology gives us *403* upgradeable proxy contracts that follow Universal Upgradeable Proxy Pattern and also *352* unique beacon proxy contracts.

At the end we find *8,225* unique upgradeable proxy contracts with unique implementation contract. We randomly sampled 150 contracts from these contracts and manually checked them, and all of them were upgradeable proxy contracts. On the other hand, we sampled 150 contracts from those marked as non-upgradeable and checked them manually. Between these 150 contracts, just two were upgradeable, and it was a false-negative. Our model did not catch these contracts because a failure happened on the decompiler to decompile the implementation contract code, so our assignment checker detector could not catch them. The reason is that implementation contracts are huge codes in contrast with the proxy contract itself.

5 Finding Admin of the Proxies

This section proposes a novel way to find the admin of the proxy contract (the agent responsible for upgrading the proxy contract) and classify them based on their account type. We apply the method to the dataset we provided from the previous analysis. We also shed light on the risks regarding the number of

[7] `opensea.io`

decision-makers who have the authority to change the whole logic of the Dapps that are using a proxy contract.

The question we will answer in this part is who can upgrade the system? There should be an agent who decides on the upgrades of the system. Generally, there are three main types of admins for upgradeable smart contracts; a single Ethereum address (a.k.a EOA), Multi-Signature wallets, and Governance schemes. EOA and multi-signature schemes are adding Centralization to the system because a limited number of private keys can change the system's whole logic. Based on *The State of DeFi Security 2021* [4] report by Certik company [8], **Centralization** is the most common attack vector of the hacked DeFi projects, so remaining a centralization point by permitting to change the system to some specific critical roles can be hazardous. We are describing three types of admins below:

Externally owned Address(EOA) This most centralized way to deal with admin is to have just one private key control the upgrades. Using EOA as admin is the fastest way to respond to incidents, but in case of a malicious admin or a private key compromise, the whole funds are at risk.

Multi-Signature Wallet A *m out of n* Multi-signature wallet is a smart contract that can execute a transaction only if m number out of a specified n EOAs agree and sign the transaction. Using multi-sig as admin is a better answer to the upgrade's decision-making than using an EOA in case of centrality while keeping the speed of an upgrade process. However, it is not decentralized. The problem here is that the Ethereum accounts are pseudo-anonymous, and the identity behind addresses is not recognizable. So, the malicious developer team can keep at least m signatures out of n in their hands, and in the desired time, they can upgrade a system to a malicious version and steal the funds. Also, it is good to mention that there are some types of governance voting which is known by *Off-chain Governance Schemes* in which users who hold governance tokens can signal their votes on proposals in an off-chain tool like *Discord* or *Snapshot* and then another agent which is a part of multi-signature schemes can put the results on-chain and execute the actions if needed. We consider the off-chain governance in the Multi-sig category because, in the end, these multi-signature wallet owners are the only on-chain agents responsible for changing the system, and there is no way to enforce the signers to reflect the off-chain agreement results to the smart contract.

On-Chain Governance Voting The most decentralized way to decide on a system change is to use a decentralized voting scheme. This can be done by distributing governance voting tokens to the community, and then they can vote on a change proposal by staking their voting token. There is some critique to this method. Governance by voting has an inherent time delay to the upgrading process. This raises a problem when the system needs an instant upgrade (e.g.,

[8] `certik.com`

responding to an incident) [9] It is also not cost-efficient for the voters because all token holders must send a transaction for voting and pay a network fee. The other problem with this method is a fair distribution of the tokens. If the governance token does not distribute fairly and the majority of tokens are granted to a limited number of users, they can change the voting results to their desired outcome. This type of admins is the most decentralized solution and reduces the risk by giving the voting right to all the token holders to be a part of system upgrades and vote on the changes. However, as mentioned above, it depends on the distribution of the tokens.

5.1 Exploring Admin Types

As described above, proxy contracts may have three types of admin; EOA, Multi-Sig, and Governance Contract. In EOA and Multi-sig types, a person or a limited number of persons may decide to take control of the system. The risks regarding each admin type discussed above bring us to find the admin types of all proxy contracts we found in our first analysis.

In the previous section, we gathered 7.3k upgradeable proxy contracts. In this part, we try to find the admin of each proxy contract and recognize the type of the admin (i.e., EOA, Multi-Sig, Governance). Here we describe our methodology of finding the admin addresses and their types. The process can be divided into two main parts; Finding the admin account's address and finding the admin type (EOA, Multi-Sig, or decentralized governance).

Finding the Admin Account's Address. EIP-1967 [**?**] suggested specific arbitrary slots for upgradeable proxy contracts to store *Admin address*[10]. So, we first check this specific storage slot and if it is non-zero, the address that is saved inside it, is the admin address. However, not all proxy contracts use the EIP-1967 suggested storage slot. So, for non-EIP-1967 proxies, we proposed a way to find the storage slot in which the admin address is stored. The process is very similar to the last part. We first find the function in which the admin can change the *target address* (upgrade function). This function is critical and should only be called by the admin. It means there should be access control to check the function caller. We found this access control check, and the address that is checked inside it is the admin address.

Finding the admin type. Having the admin address, we can check if the account is an EOA by just checking if the account consists of a code or not[11]. If the contract does not contain code, the admin is an EOA. The remained admin addresses are contracts because their account keeps code. This contract can be multi-signature smart contract wallets. The most widely used multi-signature wallet is Gnosis Safe[12] wallets. We automatically checked if the code of the admin

[9] This arises the need for another mechanism to quickly fix bugs and upgrade the system in the event of incidents in conjunction with the voting process (e.g., Global shutdown in MakerDAO).

[10] Storage slot 0xb53127684a568b3173ae13b9f8a6016e243e63b6e8ee1178d6a717850b5d6103

[11] using the *eth_getCode* method for the admin address

[12] `https://gnosis-safe.io/`

address is the Gnosis wallet multi-signature, and if yes, we marked them as Multi-Signature admins. After picking Gnosis safe wallets, we manually checked %10 of the remaining addresses to find any other patterns for multi-signature wallets and found other patterns (e.g., MultiSignatureWalletWithDailyLimit, etc.) and added them to the dataset as well.

A myriad of the contracts is using another proxy contract as their admin, which is known as *Admin Proxy*. Admin proxy contract adds another layer of indirection. In this case, the real admin (owner of the admin proxy) sends their desired transaction to the admin proxy, redirected to the primary proxy, and getting executed. So, we tried to find the admin proxies among the admin addresses and then find the owner of these proxies. The owner's address is the real admin account that can upgrade these systems. Now that we have the admin address, we do the same processes as before to find the EOA and Multi-Signature types. The remained proxy contracts which are not marked as EOA or Multi-Signatures are marked as decentralized governance or not known. We add not known tag because our model has false negatives to detect the multi-signatures because some of the contracts were using undefined new patterns as their multi-signature contracts. For a detailed explanation of the methodology and implementation, check the appendix.

By applying the above methodology in our dataset, the results show that totally out of 7.3k proxy contract, **3558** are controlled by an EOA address, **988** are controlled by a multi-signature wallet, and **2924** addresses are governance controlled, or our methodology could not find their type.

The results show that a single EOA account controls %48 of the proxy contracts and %62 by an EOA or Multi-Signature wallets control. This is a significant risk to the Ethereum ecosystem because, in these contracts, one or a limited number of persons can decide to change the whole logic of the contract and take control of the funds under the custody of the contract. Bent Finance incident [5] is a real-world example of what may happen to all these proxy contracts. Bent Finance[13] is a staking and farming platform. They are using *Transparent Upgradeable Proxy* pattern in their system. The admin of the proxy was an EOA at the time of the incident. The malicious developer deployed a new implementation contract[14] in which it provides a huge amount of token to the malicious actor's address [15]. Afterward, the attacker upgraded the proxy to the malicious implementation contract, and by doing that, a considerable amount of tokens were assigned to the attacker's address. Once the balance was transferred to the attacker, they upgraded the proxy to the latest non-backdoor version to hide the exploit. Having a massive amount of the tokens, the attacker drained liquidity from Curve Finance protocol, a decentralized exchange. The same scenario may happen to all other upgradeable proxy contracts which use EOA or multi-signature wallets as their admin.

[13] `https://app.bentfinance.com/`
[14] `https://etherscan.io/address/0xb45d6c0897721bb6ffa9451c2c80f99b24b573b9`
[15] 0xd23cfffa066f81c7640e3f0dc8bb2958f7686d1f

6 discussion

References

1. Barros, G., Gallagher, P.: Eip-1822: Universal upgradeable proxy standard (uups) https://eips.ethereum.org/EIPS/eip-1822
2. of Bits Blog, T.: Contract upgrade anti-patterns https://blog.trailofbits.com/2018/09/05/contract-upgrade-anti-patterns/
3. Buterin, V.: Delegatecall forwarders: how to save 50-98contracts with the same code https://www.reddit.com/r/ethereum/comments/6c1jui/delegatecall_forwarders_how_to_save_5098_on/
4. Company, C.: The state of defi security 2021 https://www.businesswire.com/news/home/20220113005054/en/CertiK-Releases-2021-State-of-DeFi-Security-Report
5. Finance, B.: Bent update https://bentfi.medium.com/bent-update-12ae69a41dc6
6. Murray, P., Welch, N., Messerman, J.: Eip-1167: Minimal proxy contract https://eips.ethereum.org/EIPS/eip-1167

A Evaluation of different methods

In this section we compare and evaluate different methods discussed in previous section and explain the consequences regarding each method to the users and developers of Dapps.

A.1 Criteria

There are some characteristics that can help the designer to decide which method should be used on the system and add upgradeability to the Dapp. In this part we pencil out these criteria and evaluate different methods based on these criteria. In this part we describe and specify what it means that each row of our table receives a check mark (✓), double check marks (✓✓), square (⊠) or nothing.

Can replace entire logic An upgradeability method in which the upgrader is able to replace the entire logic of the system earns a check mark(✓) otherwise it receives nothing.

can replace pre-specified part of logic An upgradeability method in which the upgrader can change *just* pre-specified part of logic of the system (and not entire logic) earns a check mark(✓) otherwise it receives nothing.

Replace entire state An upgradeability method in which the upgrader can replace the entire state in newer version earns a check mark(✓) otherwise it receives nothing.

can change pre-specified state variables An upgradeability method in which the upgrader can *just* change some pre-specified state variables receives a check mark (✓) otherwise it awarded nothing.

No need to deploy a new contract In some upgradeability patterns, the upgrader needs to deploy a new smart contract in the process of upgrade which receives nothing. Upgradeability methods which do not need to deploy a new contract for the process of upgrade receive a check mark (✓).

No need to migrate state from old contract In some patterns, there is no need to collect data from the old version and push it to the new contract which receive a check mark (✓). On the other hand, patterns which required to migrate data from old version receive nothing.

No need to separate State and Logic An upgradeability pattern that does not requires separation of logic and storage contracts awarded a check mark (✓) otherwise it receives nothing.

DelegateCall opcode Risks Some of the upgradeability methods utilize *Delegate call* opcode. Using this opcode bring complexity to the system and needs more security considerations. These security considerations are categorized into two main risks which we explained before; function selector clashes and storage clashes.

Function Selector Clashes Risk Upgradeability methods in which the developer should take care of function selector risks receive check mark (✓), otherwise receive nothing.

Storage Clashes Risk Upgradeability methods in which the developer should take care of storage clashes risks in two contracts receive check mark (✓), and the methods that the developers must consider this risk and deal with it in more than two contracts receive double check marks (✓✓) otherwise receive nothing.

No indirection Indirection happens if the first external message need be forwarded from a contract to another. Upgradeability methods that do not need any indirections receive a check mark (✓). An upgradeability pattern that contains indirections which adds an extra gas because of adding one or more layers of indirection awarded nothing. An upgradeability method in which just a portion of its transactions (and not all transactions to the contract) need indirection receive square (⊠).

User endpoint address not changed In some upgradeability methods, after the upgrade process, users must call a new contract address to use the Dapp. It is equivalent to having 2 different Dapps at the end of the upgrade. Alice uses a DApp X which uses one of the upgradeability patterns at address A before the upgrade. After upgrade, she may be unaware that upgrade happened and use the previous address (receive check mark (✓)) or she may need to use address B instead which receive nothing.

Downtime in upgrade events Patterns which we will have a downtime of the Dapp in the upgrade event receive check mark (✓) otherwise it receives nothing.

No need to change code to add the upgrade pattern Upgradeability patterns in which the developers do not need to change any part of the original code to add the upgrade method receives nothing. The methods in which the developers do not need to change the whole code but should add a proxy contract or change just one component of the system receive square (⊠) and patterns in which the devs should change the whole code to add upgradeability receive check mark (✓).

Need to change a state variable Upgradeability patterns in which the upgrader should change a state variable on the upgrade process receive a check mark (✓) otherwise it receives nothing. Two scenario could happen in this case, changing a variable as a upgrade parameter or changing an address variable which is a pointer address in the system.

A.2 Consequences

In this section we discuss about the consequence of each upgrade methods regarding the criteria we mentioned in the previous part in users and developers that want to use the upgradeability pattern or uses a Dapp that uses one of the mentioned patterns.

A.3 Speed of an Upgrade

Upgrade events of a Dapp consists of two different processes. First a way to come to an agreement to a change, and then a way to implement and execute the change. The First part depends on the reason behind the upgrade. If the upgrade is to patch a bug, then the process to come into agreement is very fast but if the goal behind upgrade is to add new functionality or change a logic, it usually starts with a proposal and after the discussion if the agent that responsible for the decision agree with the proposal, the execution part will be started. We won't discuss about the first process because it depends on the type of agent that is responsible for the upgrades. The types of agents will be discussed in details in further sections which are EOA, Multi-sig and Decentralized Governance Voting system.

After coming into agreement about the change, the speed that the upgrader can implement and execute the upgrade depends on three main criteria discussed above; *Need to migrate state from old contract* , *No need to migrate state from old contract*, and *having a downtime in the upgrade process*.

Parameter change method is the fastest way to execute the upgrade because there is no need to deploy a new contract, and no need to migrate state and no downtime in the system. *Component change* method change is not as fast as Parameter change method but faster than other types because the upgrader needs to deploy a specific smart contract which is a small component of the system and also update an address variable inside the main contract that points to that specific component and change it to the address of the new version of that component. But there is no need to migrate data and there is no downtime needed for this upgrade method.

Migration method has a slow upgrade process. The reason is that the upgrader needs to deploy a new contract and also the upgrader or users should transfer the data from old version to the newer version. In most Migration processes the developer team deploy a *Migrator* contract and users should use this Migrator contract to withdraw their funds/data from the previous version and move it to the newer version. But, there is no downtime in the Dapp and no need to change a state variable.

Call-based and *DelegateCall-based* are very similar to each other in the speed of upgrade. These two are not as quick as *Retail changes* because the developer needs to implement and deploy the *whole logic* contract to the blockchain and then change the pointer addresses inside the storage/proxy contract to the newer version. *Diamonds* is very similar to these two but because Diamonds is a modularized pattern, in the event of upgrade we need to just implement and deploy one module which is related to the functions we want to change. So, the speed in Diamonds is similar to Component change methods. On the other hand these two approaches are faster than *Migration* because as mentioned before, there is no need to migrate data. There is no downtime in these methods.

Metamorphic method is the slowest way to upgrade a system which uses this method because similar to the Migration plan there is a need to deploy a contract and migrate the state to the newer version but there is a difference between these two. In Metamorphic method the upgrader first should *Self-Destruct* the previous version in a single transaction and after that transaction send a contract creation transaction to deploy the newer version. Because self destruct happened at the end of the transaction, the process of upgrade happens on two different transactions which is a downtime to the system. This downtime could be a gap between order of the two transaction in a single block or could be gap between blocks that these two transactions included into blockchain.

A.4 Cost of Upgrade

One of the main differences between upgradeability approaches is how much does the upgrade process costs for the upgrader and users. The cost of upgrade mostly depends on three criteria explained above; need to deploy a new contract, need to migrate a the state to newer version, and need to change a state variable.

Parameter change method is the cheapest method in the upgrade event because there is no need to deploy a new contract or migrate data but just need to change a state variable.

Component change is in the middle because there is a need to deploy a new contract (however it is cheaper comparing to methods in which we should deploy the whole logic), and change an address pointer variable but there is no need for data migration.

Migration plan is very expensive in the upgrade event because we need to deploy a new contract and migrate the data from the old version which is very expensive. But no need to change any state variables.

Call-based and *DelegateCall-based* are very similar to each other in the cost of upgrade which is more expensive than component change but cheaper than migration. In both the upgrader must deploy the a contract containing the whole logic and change an address pointer inside storage/proxy contract. But there is no need to migrate the whole data.

Diamond's cost of upgrade depends on the upgraded needed for the system. It is very similar to Delegate-based pattern but if there is a need to change some functions that are not in one module (faucet) of the system then we need to deploy more than one smart contract in the event of upgrade and so it is more

expensive than doing upgrade comparing to Delegatecall-based pattern (however we do not need to deploy the whole logic but deploying a contract to ethereum blockchain is the most expensive action we have in EVM).

Metamorphic method is the most expensive method we have because we need to deploy a new contract, migrate data to the newer version and also we need to self destruct the previous version before the upgrade event which adds cost to the upgrade process.

A.5 Gas overhead for users

Sometimes in upgradeability patterns, we have a tradeoff between adding a feature to the pattern to improve it and increasing the cost for users that want to interact with our Dapp.

In patterns that needs indirection, such as *Call-based*, *Delegatecall-based*, *Diamonds* and *Component change* pattern we are adding a cost to the users because for all or some of the transactions to the Dapp, our system needs to forward the calls to another contract using Call or Delegatecall opcode to the users. Also in *Delegatecall-based* and *Diamond* pattern to mitigate the function selector clashes or storage clashes we need to add some checks to our code which also increases the cost of interacting with the Dapp. We can compare all patterns like UUPS or transparent proxies in term of gas overhead here.

Also there are some other ideas that addresses some limitations of a upgradeability pattern but increases the cost for users. For instance in *Call-based* approach one of the problems is that after upgrade users should use a new address for using the Dapp but adding a *Registry* contract can help to mitigate this. Using Registry contract, all other contracts should ask the registry to find out the latest version of the contract and then calls to the newer version which adds a gas cost to the users.

A.6 Useability

Upgradeability patterns differ in term of Useability and it depends on three criteria explained above; *User endpoint address changed*, *Need to migrate state from old contract* and *Downtime in upgrade events*.

Patterns in which the endpoint address is changing after upgrade event, *Migration* and *Call-based* is not user friendly because each time that the upgrade happened, the user must use the newer address. So make awareness about theis change is a hard action and need to socially interact with the users and make them aware of the change. We have two main type of users in the Dapp ecosystem, normal user or another smart contract (Dapp) that uses our system. Regular users which uses the official interface (website) of the project may do not sense any changes but users that work with the smart contract directly or via their own interface or other Dapps that uses the smart contract must have a way to upgrade the address they uses to use the newer version and if they did not implement a way to upgrade this address then their Dapp will face problems. So these patterns are make problems for composablility of the ecosystem.

In *Migration* plan, in most cases of upgrade events the users are responsible for the migration of data. For instance, the user must withdraw the fund and use a *Migrator* contract to push the data into the newer version which add costs to the user and it is not user friendly. This is one reason that make the Migration plans very hard because some users are not doing the process of migration and stay on the previous version which is like having a fork for the Dapp in side of the Dapp team. We see this happened on Uniswap V2 and V3.

In *Metamorphic* pattern as mentioned before there is a downtime during the upgrade. So users cannot work with the Dapp on that exact time which is not user friendly.

A.7 Dealing with two different new versions

In *Migration* and *Call-based* pattern we will come up with two different Dapps. So a decision must be made for the previous version. One possible choice could be shutting down the old version. It can be done by self-destructing the old version, or by pausing mechanism which will be explained in further sections. In migration plan it is not regular to stop the previous version because in most migration plans, users are responsible to move their funds and data from the previous version to the new one and we cannot force them to do that, so we cannot stop the smart contract.

The other option could be having a mechanism that after the upgrade, all calls to the previous version just be forwarded to the newer version which add costs and have some limitations like we cannot call the new functions defined in the newer version using the old version. This option is doable in Call based patterns. The other problem of this option is that if we upgrade a system more than one time then the calls to the first version should be indirected through lots of contracts to reach to the newer version. Also it adds complexity because developers must maintain more than one contract [2].

A.8 System Complexity

Using upgradeability patterns will add to complexity of our system but the degree of complexity varies and depends on the pattern. *Parameter Change* method does not change the system in general but just adding a mechanism to change pre-specified variables in the system. The most important issue about the Parameter change method is that the developer team must limit the boundary of these parameter for the security of the system. For instance in MakerDao platform, Stability fee is changeable but if this variable be changed to %100 then the whole system will be halted.

Component Change pattern is very similar to the Parameter change but here a whole component could be changed and finding the safe boundary of changes and limiting this boundary is a bit harder.

Migration plans for upgradeability does not change any complexity to the system because we do not need to change any part of system to add this type of upgradeability to it. The only important issue regarding this pattern is that we

must be sure that there is a way to collect data from the old version like having getter functions for reading data and also having a withdraw function for users to collect data and funds from previous version and push or deposit it to the newer version.

Using *Call-based* patterns adds higher degree of complexity to the system compared to previous patterns. As discussed before in this pattern we must be sure that the storage and logic contract is divided and there is not any storage variable inside the logic contract. This is one of the main security issues that found in the Dapps using this pattern regarding Trail of Bits company reports [2]. As mentioned before to add a way to storage contract to define new variables, developers uses the Eternal Storage pattern for their storage contract which is very hard to apply for complex data structures in Ethereum such as mappings or structures. This is another source of complexity using Call-based pattern.

Delegate-call pattern adds complexity to the code because of using *Delegate-call* opcode in its logic. As mentioned above because of using this opcode, the developer should take care of storage clashes and also function selector clashes. Other than these two there are some other limitations and risks of using this patterns. For instance, we cannot have a *Constructor* function on the logic contract (implementation contract) because constructor functions is used to initialize specific variables at deployment time and if we have a constructor inside the logic, then storage of implementation contract will be changed and not storage of proxy contract. To mitigate this problem we can add a regular function named *Initialize* function inside the implementation and make sure that this function can be called *once* to act just like a constructor function. Using initialize function brings some security risks that we will explain in further sections.

Diamonds pattern is very similar to the Delegate-call based patterns and have the same risks but this pattern is more complex because here we have different implementation contract for a single proxy and we should be sure that all of these implementation contracts share the same storage layout otherwise we will have a storage clash problem.

Metamorphic pattern is proposed recently and not well-tested yet. There are some risks to this pattern as well. We should be sure that we have a mechanism to self-destruct the contract. Otherwise if we cannot self-destruct the current contract we cannot redeploy a new version and so our contract won't be upgradeable. The other important issue related to Metamorphic pattern is that the developer must know that each time they want to upgrade the system the whole storage will be wiped out and need to re-initiate the whole state after re-deployment.

B Implementation detail of Finding The proxy contracts

Implementation In this section we describe the processes of finding upgradeable proxy contracts explained in the previous section in detail of implementation and execution. Each block of Ethereum blockchain consists of transactions that processed in that exact block. To get all transaction details we need to replay the

transaction and collect the data of execution of the transaction. Ethereum full archival node has a method, *trace_transaction*, that gives the *Parity VM transaction traces* of the transactions executed on the specific block. This transaction traces are composed of *actions*. Each action gives data about each specific part of the execution of the transaction and consists of the opcode instruction that is executed on that action and the input data relevant to the that instruction and also information about how it manipulates the state, memory or stack of the Ethereum Virtual Machine (EVM).

Having the transaction traces for each block, we will search to find actions that consists of *callType*. When an action has callType field it means that there is a call happened in that specific action, which means one of *Call,Static Call* or *Delegatecall* happened in that action. we picked actions that have *callType* and this call type is *delegate call*. Also each action has an *input* element for messages that shows the call data (the data that is used for the calling the callee address). If the input element of the selected actions are equal to the input element of their previous action with the same transaction hash, it means the transaction passes a fallback function which delegate calls it to another contract. Because the input is not changed after the call it means that it passes a fallback function because the function selector (first four byes of the input) has not been changed which means the called function does not exists in the contract, so the fallback is called. Also the delegate call in the action shows that inside that fallback function there is a delegatecall that passes the whole message data to the target address contract. So, these contracts meet first condition described in the methodology part and these contracts are proxy contracts. We collect *From address*,*To address* and the *transaction hash* of these picked actions and pick the unique from addresses of this data. From address is the address of sender of transaction (proxy contract addresses), to address is the address which the proxy sends the transaction into (implementation contract addresses) and transaction hash is the transaction identifier that is used in Ethereum blockchain.

As discussed before, these selected contracts are proxy contracts and not necessarily upgradeable proxy contracts. So, we need to filter the *forwarders* from proxy contracts to find upgradeable proxy contracts. As mentioned in methodology section, the upgradeable proxy contracts must satisfy a condition; the admin of the contract should be able to change the target address. In other words, we should check if the target address inside the proxy contract is changeable or not.

There are five regular ways for the situation that the target address is fixed and not changeable on the contract:

1. The target address is hardcoded in the contract without assignment to any variables
2. The target address is saved in a constant variable type
3. The target address is saved in an immutable variable type and the deployer sets it via a constructor function
4. The target address is defined in a storage variable, but there is no function or a way to change this address after contract deployment (neither in proxy contract nor in implementation contract)

5. The proxy contract grabs the target address each time by calling another contract and there is no way on the callee contract to change this address

In the first three situations, the target address amount will be appeared on the runtime bytecode (the bytecode that is saved in the blockchain after deployment) of the smart contract. We can get the bytecode of each proxy contract that are collected from previous part using the *eth_getCode* method of a Ethereum archival node and check whether the target address is appeared on the bytecode or not. To have the target addresses we just need to collect The *To addresses* that we collected on the previous part which are supposed to be the implementation addresses because the proxy contract delegate calls into these addresses. So we check if the target address is appeared on the bytecode of the proxy contract. If yes, these proxies are not upgradeable proxies and they are forwarder contracts because the address is hardcoded on the bytecode and cannot be changed after deployment. Finding the forwarders, we removed them from the list of the proxy contracts.

For the situation 4, we need other processes to check whether the implementation address is changeable or not. We need to design a filter to tell us if there is a way to change the target address after deployment or not. We will describe the filter in the later part of the paper, but for now let's assume that we have a module named *Assignment Checker* that checks the code and tells us that whether there is a function inside the contract that the caller can change the *target address* using the function. If yes we mark the contract as an upgradeable proxy contract because there is a way in the proxy contract to change the target address and point it to the new implementation contract and upgrade the system.

For the situation 5, we check the proxy contract to find whether each time before the delegate call, the target address inside the proxy contract is coming from an external call to another contract. If yes the we check the callee smart contract (a.k.a Beacon contract) that if the target address is changeable inside that contract or not. If it is changeable we mark the proxy as *Beacon Proxy* otherwise it is a *Forwarder*.

But this is not the end of story. There is another proposed upgradeable proxy contract pattern, named Universal Upgradeable Proxy Standard (UUPS) also known as EIP-1822 [1] that described in the previous sessions. In this type of proxy contracts, the target address can be changed using the implementation contract and not the proxy itself. As mentioned before the code of the implementation contract is executed in the context of the proxy contract and so it will change the storage state of the proxy contract. If we have a function in the implementation contract that gives us the ability to change the storage slot of the target address, we can upgrade the system by calling that function of implementation contract through proxy itself. We couldn't catch this type of upgradeable contracts by the previous processes because we just checked if there is a function inside the proxy contract itself that can change the target address.

We can tackle this problem by first finding the storage slot of the target address inside proxy contract. Then by using the *Assignment Module* to check if

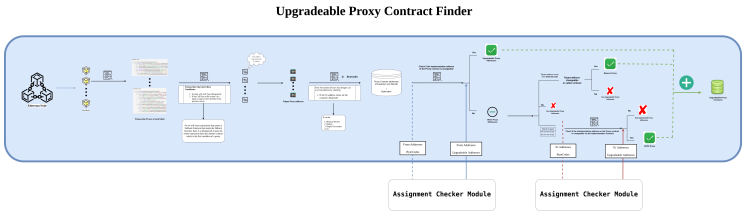

Fig. 4. Upgradeability Proxy Contract Finder

there is a function inside implementation contract which gives the admin right to change the target address. So first we should find the storage slot of the proxy contract in which the target address is saved. There are two EIPs out there that makes it easy to find these storage slots that suggested to be used to save target address for upgradeable proxies; EIP-1967 [16] and EIP-1822 [17], suggested randomly selected storage slots for target address to mitigate the overwriting on these slots. The UUPS contracts are usually using one of these two storage slots to save their target address.

The more general way to do this is not using these proposed slots from these EIPs for this process. The general way is to find the storage slot from the proxy contract and then check if admin can change this specific storage slot inside the implementation contract. But this way is not doable in large scale because in *Assignment checker* module we need to decompile the bytecode which is time consuming for contracts that have a large bytecode. Implementation contracts are usually large pieces of codes. So, it is not possible to do this process in a large-scale for all proxy contracts that are found from the previous parts. This is the reason that we limit our filter to specific storage slots proposed in EIP 1967 and EIP 1822.

So for this part we use the remained proxy contract addresses from the previous part. First we select the contracts that their target address is saved on the mentioned storage slots and if the target address is saved on the slot, check the implementation contract to find variable regarding that storage slot defined in the implementation contract. Afterwards check if the variable is changeable in a function in implementation contract. If it is changeable, it means that the admin can upgrade the system using a function on the implementation contract, and so the proxy is an UUPS contract.

The whole process is depicted on figure B.

Assignment Checker Module As mentioned in the previous part, we need a module to check whether the admin can change target address on the proxy contract,using a function in the proxy contract, implementation contract or beacon contract. For this purpose the module must get the *Bytecode* of the

[16] 0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc
[17] 0xc5f16f0fcc639fa48a6947836d9850f504798523bf8c9a3a87d5876cf622bcf7

proxy,implementation or beacon address as input and find the variable name and also its storage slot of the target address. Then checks to find out is there any function inside the contract that gives the admin the ability to change the target address.

We use bytecode decompiler named *Panoramix decompiler* [18] to decompile the bytecode into well-formatted python language codes. The decompiled code gives us all storage variables of the related contract and the storage slots of those variables in a function named **Storage**. On the other hand, the decompiled code will tell us if a function is *Payable* or not. Among these Payable functions the one that does not have name or its name is fallback is the *fallback* function of the contract. So we will try to find the line of code that *Delegate Call* happened on it and collect these lines. Now that we have storage variable names and storage slots of these variables and also the line of code inside fallback that have the delegatecall, we will check to find the target address variables. We are doing that by checking if one of the storage variables inside Storage function is used in the line of code that contains delegate call. We will add them to an array of implementation addresses.

There is two other steps here. First finding other variable names with the same storage slots as the implementation addresses we found from the first step by checking the Storage function and also finding another variables that being assigned to those implementation variables in some other part of the code. We will add these two type of variables to the implementation addresses as well.

Now that we have a list for implementation addresses, we will search through the code to find if any assignment happened to one of them. If yes we will pick the variables that is assigned to target variable and then check if this assignment happened in a specific function and to one of the inputs of that function. In this case this function will be the upgrade function because the caller of this function can upgrade the target address by calling this function with desired input.

To summarize what we did, we find all possible variables in the code that can change the target address inside the contract and check if there is any function inside them that can assign new address to the target address variable.

The whole process is depicted on figure B.

B.1 Detailed Explanation of finding Admin types

The whole process is depicted in the figureB.1.

EIP-1967. As mentioned above EIP-1967 [**?**] suggested specific arbitrary slots for upgradeable proxy contracts to store implementation contract's address and *Admin address*[19].

In first step we use *eth_getStorageAt* method of an Ethereum full archival node to search the EIP-1967 specified storage slot for admins on our 7.3k proxy

[18] `https://github.com/palkeo/panoramix`

[19] Storage slot 0xb53127684a568b3173ae13b9f8a6016e243e63b6e8ee1178d6a717850b5d6103 for admin

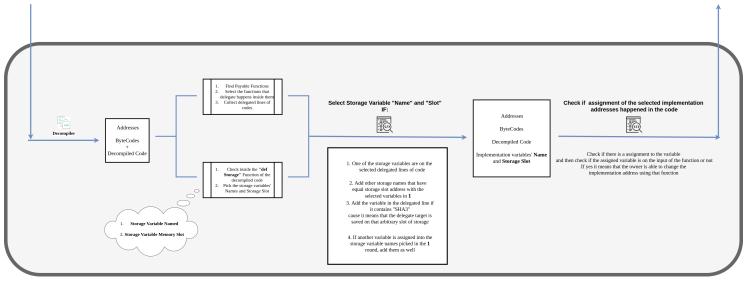

Fig. 5. Assignment CheckerModule

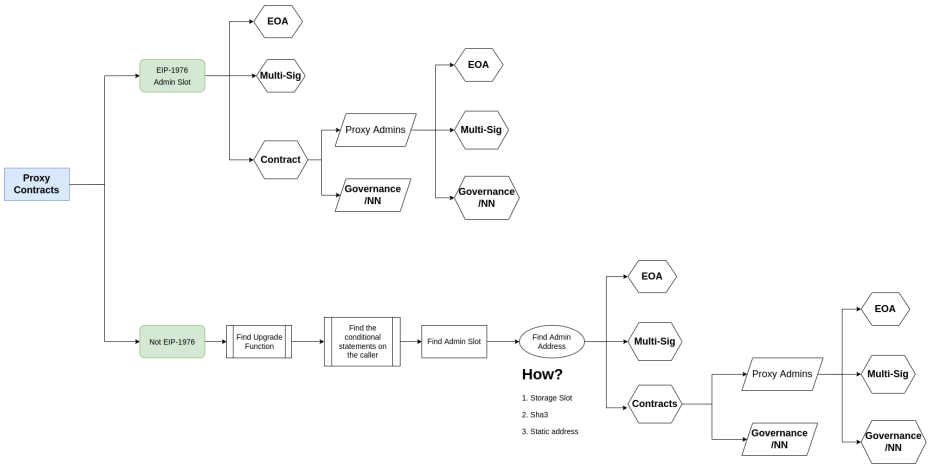

Fig. 6. Admin Types

contracts. If the result of this method is non-zero it means that the proxy uses EIP-1967 standard because the specified storage slot is an arbitrary slot and one can store variable in this slot just by defining this slot which means that they used EIP-1967.

So, for non zero results, we capture the address which is the address of admin of the proxy. Now we try to find the type of these admin addresses. Having the address of the admin we use *eth_getCode* method to check the code of the admin account. If the code is empty, it means that this account is not a smart contract so it is an EOA. we find *900* EOA admins that their proxy uses EIP-1967 standard.

The remained admin addresses are contract because their account keeps code. This contract can be multi signature smart contract wallets. The most widely used multi signature wallet is Gnosis Safe[20] wallets. We automatically checked if the code of the admin address is the Gnosis wallet multi signature patterns.

[20] `https://gnosis-safe.io/`

After picking Gnosis safe wallets we manually checked %10 of the remained addresses to find if they used other patterns for their multi signature wallet and we found some other patterns (e.g. MultiSignatureWalletWithDailyLimit, etc.). After Finding all these types we checked the admin codes to see whether they are multi signature wallets. We find *255* admin accounts that uses multi signature wallets as their admin.

There is another class of admin contracts named *Admin Proxy* contracts. These admin proxy contracts are another layer of re-direction between the real admin and the Dapp's proxy contract. The admin proxy contracts are proxy contracts that redirect the messages from the real admin into the Dapp's proxy. The only person who can use admin proxy is the admin (a.k.a owner) of the admin proxy. So we first filter the admin proxy contracts using the codes we get from the previous part and then try to find the owner of the admin proxy contracts. The owner of admin proxy contract (the real admin) also can be EOA, Multi-sig or governance contract. Finding the owner of the admin proxy contract, we used *eth_getCode* method to check the code of these account and find out if they are EOAs or Multi-signatures or governance schemes. Doing this we find *1202* EOA admin accounts and *567* multi signature admins. We marked the remained proxy admin addresses as Governance/Not Known admin types and we have *462* of them. There were also non admin proxy contracts which use EIP-1967 but they were not EOA or Multi signatures. We marked them as Governance/Not Known admin types and we have *53* of them.

Non EIP-1967. For proxy contracts which not use EIP-1967, the problem is we don't know where the admin address is saved in the proxy contract's storage (what is the storage slot of the admin address). It can be saved in a storage slot of the contract or be hardcoded in the smart contract[21].

So there are two ways that the admin address is saved in the proxy contract. It can be saved as a storage variables or it can be hardcoded as a fixed address.

In storage variable case, the first question is in which storage slot the admin address is stored. So, the first step is to find the storage slot of the admin address variable. Also for the fixed address we should find the fixed address of the admin directly.

To find the slot of the storage variable in which admin address is saved, we first find the function in which the proxy can be upgraded. For finding the upgrade function we exactly do what we did in B part. We first find the storage variable in which we saved the implementation address and then we find a function in which the implementation address can be changed using the inputs of that specific function.

The upgrade function of a proxy contract is a critical function and the only account that can call this function should be the admin of the proxy contract. So, there should be an access control check inside the upgrade function to check

[21] There are some other possible ways to store the admin address for instance saving it in another contract and each time make an external call to get the address but to our knowledge this pattern is not widely used as a standard

whether transaction sender is equal to the admin address or not. So, after finding the upgrade function we search for conditionals that checks the caller of the transaction and by doing that we can find the admin address or the storage variable in which the admin address is stored.

If the admin address is stored in a storage variable, then we should find the storage slot of that specific storage variable. For finding the storage slot we do what we did in B part by using *def storage* function of the decompiler and check the storage slot of the storage variable we found, and the admin address is saved on it. Now we have the storage slot of the admin address and we should start doing all the things we did for EIP-1967 in the previous part. In the EIP-1967 the storage slot for admin address was pre-specified and we do not need to find the slot but in this case we use the above methodology to find the slot but the further steps are the same as EIP-1967. So, by using the *eth_getCode* method for admin address inside the storage slot we find above, we can check wether the admin is EOA, Multi-sig, Governance, Proxy admin or not known. In this part we find *1313* EOA addresses and *104* multi-sig admins. Also by checking proxy admins we find *92* EOA addresses and *16* Multi signatures that uses proxy admin as a level of indirection.

In another case the admin address may be stored directly in a specific arbitrary storage slots. In this type the compiler will specify the address using the *sha3* hash notation. In this case same as above we find the conditional check on the transaction sender and then find the storage slot in that line and hash of that pre-specified string. By finding this arbitrary storage slot and doing the same processes we did in the previous part we find *2* EOA addresses and *10* Multi-sig addresses.

The only case that is left is proxy contracts, in which the address of the admin is hardcoded inside them. It very straight forward. We find the upgrade function and the access control check on the caller of the transaction and then pick the fixed admin address and do the same processes mentioned above to find the admin types. There are *49* EOAs, *36* multi-signature admins and *160* governance and not known admin addresses.

So, totally out of 7.3k proxy contract, **3558** are controlled by an EOA address, **988** are controlled by a multi signature wallet and **2924** addresses are governance controlled or our methodology could not find their type.