Not so immutable: Upgradeability of Smart Contracts on Ethereum

Mehdi Salehi (✉), Jeremy Clark, and Mohammad Mannan

Concordia University, Montreal, Canada
salehi.mehdi70@gmail.com

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 upgradeability, raising the research questions of *how* upgradeability is achieved and *who* is authorized to make changes. In this paper, we summarize and evaluate six upgradeability patterns. We develop a measurement framework for finding how many upgradeable contracts are on Ethereum that use certain prominent upgrade patters. We find 1.4 million proxy contracts which *8,225* of them are unique upgradeable proxy contracts. We also measure how they implement access control over their upgradeability: about 50% are controlled by a single Externally Owned Address (EOA), and about 14% are controlled by multi-signature wallets in which a limited number of persons can change the whole logic of the contract.

1 Introductory Remarks

The key promise of a smart contract running on Ethereum is that its code will execute exactly as it is written, and the code that is written can never be changed. While Ethereum cannot maintain this promise unconditionally, its assumptions (*e.g.,* cryptographic primitives are secure and 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 behavior. Unfortunately, disguising malicious behavior in innocuous-looking code is possible ('rug pulls'), and many blockchain users have been victims. On the other hand, if the smart contract is long-standing with lots of attention, and security assessments from third-party professional auditors, the immutability of the code can add confidence.

The flip-side of immutability is that it prevents software updates. Consider the case where a security vulnerability in the code of a smart contract is discovered. Less urgently, some software projects may want to roll out new features, which is also blocked by immutability. There is an intense debate about whether this is a positive or negative, with many claiming that 'upgradeability is a bug.' [1]

[1] "Upgradeability Is a Bug", Steve Marx, Medium, Feb 2019.

We do not take a position on this debate. We note that upgradeability 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 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 be 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 issue 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 up a contract logic and state, utilizing Ethereum-specific tricks (fallback functions to capture unexpected function names), and trying to reduce the gas costs of indirection between contracts.

Contributions and Related Work. The state of smart contract upgradeability methods in Ethereum is mainly discussed in non-academic, technical blog posts [13,2]. In Section 2, we systemize the different types using these resources, and provide a novel evaluation framework for comparing them.

Fröwis and Böhme [8] conducted a measurement study on the use-cases of the `CREATE2` opcode in Ethereum blockchain, which one of them is the Metamorphosis upgradeability pattern discussed in Section 2.5. They also find, in a passing footnote, some delegate-call based contracts by assuming compliance with the standards: EIP-897, EIP-1167, EIP-1822, and EIP-1967. In our paper, we contribute a more general pattern-based measurement that is not specific to a standard or a commonly-used implementation. We also are the first, to our knowledge, to study who is authorized for upgrading upgradeable contracts, shedding light on the risks of different admin types.

Recent papers have provided security tools for developers that compose with upgradeablity patterns based on `DELEGATECALL` [18,14]. Numerous measurement studies have used Ethereum blockchain data but concern aspects other than upgradeability [15,6,17,20,16,9]. Chen et al. [5] survey use-cases of the *SELFDESTRUCT* opcode, but they do not cover how it is used in Metamorphosis 2.5.

2 Classification of Upgrade Patterns

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

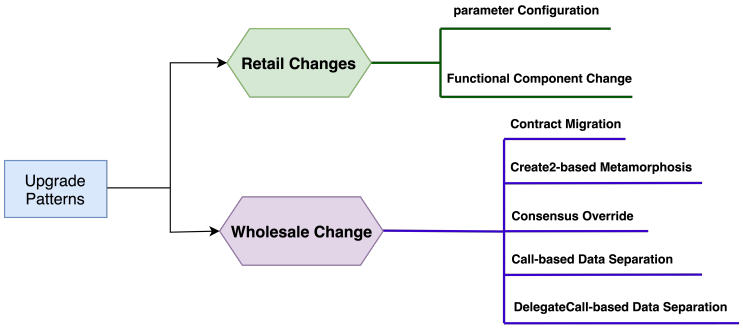

Fig. 1. Classification of upgradeability patterns.

capabilities. We only use the term upgrade and distinguish between retail (parameters and isolated code) and wholesale (entire application) changes to a smart contract. While upgrades to a smart contract's user interface (UI) can significantly change a user experience and expose new features, UIs 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.

A variety of upgradeability patterns have been proposed for smart contracts. Most leverage Ethereum-specific operations and memory layouts and are not applicable to other blockchain systems.

2.1 Parameter Configuration

We first categorize 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 4, we dive deeper into the question who can upgrade a contract.

2.2 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 contract is stored as a variable in the primary contract. Whenever this 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 [2] uses this pattern for their interest rate models [3] which are tailored specifically for each asset. The model for one asset can be changed without impacting the rest of the contract [13].

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 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 (Sections 2.6 and 2.7).

2.3 Consensus Override

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 first 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.

[2] `https://compound.finance`

[3] `https://github.com/compound-finance/compound-protocol/blob/v2.3/contracts/InterestRateModel.sol`

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 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 Foundation 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.

2.4 Contract Migration

The simplest wholesale upgrade pattern is to deploy a new version of the contract at a new address, and then inform users to use the new version—called a 'social upgrade.' One example is Uniswap[4], 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 any new logic to support upgradeability, as in many of the remaining patterns, which can ease auditability and gas costs for using the contract. However for most applications, there will be a need to transfer the data stored in the old contract to the new one. This is generally done in one of two ways. The first is to collect the state of the old contract off-chain and load it into the new contract (*e.g.,* via its constructor). 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, specific to certain applications like tracking a user's balance of tokens, is to have the user initiate (and pay the gas) for a transfer of their balance to the new contract.

2.5 `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[5] `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 this and know in advance what address will be assigned to contract A, invariant

[4] `https://uniswap.org`

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

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.[6] 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.[7] 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. `SELFDESTRUCT` opcode wipes out the contract's code and storage of the contract account that executes the `SELFDESTRUCT` opcode. Mechanically, the consequences of `SELFDESTRUCT` on the EVM are only realized at the end of the transaction. In a followup transaction, the developer 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 contract 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.

2.6 `CALL`-based Data Separation

To avoid migrating the stored data from an old contract to an upgraded contract, a contract could instead store all of its data in an external "storage" contract. In this pattern, calls are made to a "logic" contract which implements the function (or reverts if the function is not defined). Whenever the logic contract needs

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

[7] "The Promise and the Peril of Metamorphic Contracts." 0age, Medium, Feb 2019.

to read or write data, it will call the storage contract using setter/getter (aka accessor/mutator) functions. An upgrade consists of (1) deploying a new logic contract, (2) pausing the storage contract, (3) granting the new logic contract access to the storage contract, (4) revoking access from the old contract, and (5) unpausing the storage contract.

An important consideration is that the layout of the storage contract cannot be changed after deployment (*e.g.,* we cannot add a new state variable). This can be side-stepped to some extent by implementing a mapping (key-value pair) for each primitive data type. For example, a new uint state variable can be a new entry in the mapping for uints. This is called the Eternal Storage pattern (ERC930). It however requires that every data type be known in advance, and is challenging to use with complex types (*e.g.,* structs and mappings themselves).

A variant of this pattern can introduce a third kind of contract, called a proxy contract, to address the social upgrade problem. In this variant, users permanently use the address of the proxy contract and always make function calls to it. The proxy contract stores a pointer (that can be updated) to the most current logic contract, and asks the logic contract to run the function using `CALL`. Unlike the functional component pattern (Section 2.2), the proxy will catch and forward *any* function (including new functions deployed in updated logic contracts) using its fallback function. With or without proxies, this pattern is very powerful, but instrumenting a contract to use it requires deep-seated changes to the contract code. As our measurements will show, it has fallen out of favour for the cleaner `DELEGATECALL`-based pattern (Section 2.7) that addresses the same issues with simpler instrumentation.

2.7 `DELEGATECALL`-based Data Separation

This pattern is a variant on the idea of chaining each function call through a sequence of three contracts: proxy, logic, and storage. The first modification is reversing the sequence of the logic and storage contracts: a function call is handled by the proxy which forwards it to the storage contract (instead of the logic contract). The storage contract then forwards it to the logic contract using `DELEGATECALL` which fetches the code of the function from the logic contract but (unlike `CALL`) runs it in the context of the contract making the call—*i.e.,* the storage contract. When upgrading, a new logic contract is deployed, the proxy still points to the same storage contract, and the storage contract points to the new logic contract. Since the proxy and storage contracts interact directly and are both permanent, the functionality of both can be combined into a single contract. It is common for developers to call this the 'proxy contract,' despite it being a combination of a proxy and a storage contract.

This pattern is generally cleaner than using the previous `CALL`-based pattern because the logic contract does not need any instrumentation added to it. It is an exact copy of what the contract would look like if the upgrade pattern was not being used at all. However this does not mean the pattern in a turn-key solution. Each new logic contract needs to be programmed to respect the existing memory layout of the storage contract, which has evolved over the use of

all the previous logic contracts. The logic contract also needs to be aware of any functions implemented by the storage contract itself—if the same function exists in both the storage contract and the logic contract (called a function clash), the storage function will take precedence.

The main issue with function clashes is that the proxy contract needs, at the very least, to provide an admin (or set of authorized parties) the ability to change the address of the logic contract it delegates to. This can be addressed in four main ways:

1. Developers are diligent that no function signature in the logic contract is equal to the signature of the upgrade function in the proxy contract (note that signatures incorporate a truncated hash of the function name, along with the parameters types, so collisions are possible).
2. As found in the *universal upgradeable proxy standard (UUPS)* (EIP-1822): implement the upgrade function in the logic contract, which will run in the context of the proxy contract. Its exact function signature must be hard-coded into the proxy contract. Every logic contract update must include it or further updates are impossible.
3. As found in the *beacon proxy* pattern (EIP-1967): deploy another contract, called the beacon contract, to hold the address of the logic contract and implement the setter function for it. The proxy contract will get the logic contract address from the beacon every time it does a `DELEGATECALL`. The admin calls the beacon contract to upgrade the logic contract, while normal users call the proxy contract to use the DApp.
4. As found in the *transparent proxy* pattern (EIP-1538): inspect who is calling the proxy contract (using `msg.sender()`)—if it is the admin, the proxy contract catches the function call and if it is anyone else, it is passed to the proxy's fallback function for delegation to the logic contract.

A drawback of the entire `DELEGATECALL`-based pattern is that logic contracts need to be aware of the storage layout of the proxy contract. In a stand-alone contract, the compiler (*e.g.,* Solidity) will allocate state variables to storage locations, and using `DELEGATECALL` does not change that, however new logic contracts need to allocate the same variables in the same order as the old contract, even if the variables are not used anymore. This can be made easier with object-oriented patterns: each new logic contract extends the old contract (inheritance-based storage). Other options include mappings for each variable type (eternal storage) or hashing into unique memory slots (unstructured storage). The *Diamond Storage* pattern (EIP-2535) breaks the logic contract into smaller clusters of one or a few functions that can be updated independently, and each can request one or more storage slots in a storage space managed by the proxy contract itself.

2.8 Evaluation Framework

Table 1 summarizes the pros and cons of each upgradeability pattern, omitting consensus override as it is only used in emergencies. Further detail and some

Method	Can replace entire logic	No need to migrate state from old contract	User endpoint address unchanged	No need to instrument source	No need to deploy a new contract to upgrade	No indirection between contracts	No downtime to upgrade	No function selector clashes	No storage clashes
Parameter Configuration		•	•	•	•	•	•	•	•
Component Change		•	•	○		○	•	•	•
Contract Migration	•			•		•	•	•	•
Create2 metamorphosis	•		•	•		•		•	•
Call-based	•	•					•	•	•
DelegateCall-based	•	•	•	○			•		

Table 1. An evaluation of upgradeability patterns. • indicates the upgrade method is awarded the benefit in the corresponding column. ○ partially awards the benefit. Empty cells shows that the method does not satisfy the property.

take-aways from the evaluation are in the appendix of the full version of this paper [19].

3 Finding Upgradeable Contracts

We now design a series of measurement studies to shed light on the prevalence of the various upgrade patterns. We exclude retail changes from our measurements, because variable changes and external function calls are too commonplace to distinguish. We focus on wholesale patterns, and devote the most effort to finding contracts using the `DELEGATECALL`-based data separation pattern (Section 2.7) as these are the most widely used and there are various sub-types (UUPS, beacon, *etc.*). The other types of wholesale patterns are:

- **Consensus override:** Only 1 occurrence to date (the DAO attack [7]).
- **Contract migration:** Not detectable in code; relies on social communication of the new address.
- **`CREATE2`-based metamorphosis.** Already measured by Frowis and Bohme [8] in a broader study of all uses of `CREATE2`. They found 41 contracts between March 2019 and July 2021 that upgraded using this pattern.
- **`CALL`-based data separation.** We conducted a quick study of 93K contracts with disclosed source code [12]. We identified the Eternal Storage pattern using regular expressions and found 140 instances, the newest having been deployed over 3.5 years ago. We conclude this pattern is too uncommon today to pursue a deeper bytecode-based on-chain measurement.

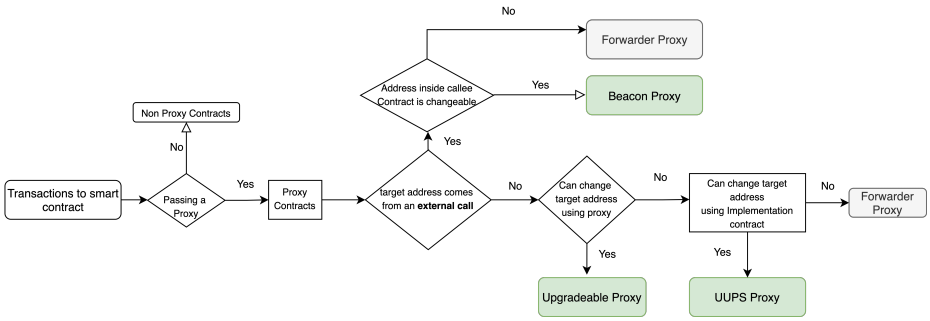

Fig. 2. Flowchart for distinguishing upgradeable contracts (green) from forwarders, and for determining the upgradeability pattern type.

3.1 Methodology

Finding proxies. While not every use of a proxy contract is for upgradeability (*e.g.,* minimal proxies [11], `DELEGATECALL` forwarders [4], *etc.*), all `DELEGATECALL`-based upgradeability variants have the functionality of a proxy. We therefore start by measuring the number of contracts with a proxy component, and then filter out the *Forwarders* which do not enable upgradeability. To identify proxies, we examine every `DELEGATECALL` action and see if it was proceeded by a call with an identical function selector to the contract making the `DELEGATECALL` action, which indicates the contract does not implement this function and instead caught it in its fallback function, and is now forwarding it to another contract at, what we will call, the *target address*. We used an Ethereum full archival node[8] and replayed each transaction in a block to obtain Parity VM transaction traces. `DELEGATECALL` is one `callType` of an `action` within a trace. Specifically, if the data of two consecutive actions of a transaction are equal and a `DELEGATECALL` is in the second action, it shows that the transaction passes the fallback function (if any other function in the contract is called, other than fallback, then the first four bytes of the data will be changed). The `DELEGATECALL` indicates the fallback transferred the whole data to the target address without altering it, which means the contract implements a proxy.

Distinguishing forwarders and upgradeability patterns. In an upgradeable contract, the target address for the `DELEGATECALL` must be modifiable. If it is fixed, we tag it as a forwarder. We define five common patterns for determining the target address cannot be changed:

1. The target address is hardcoded in the contract.
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 in a constructor function.

[8] `https://archivenode.io/`

4. The target address is defined as an unchangeable storage variable.
5. The proxy contract grabs the target address by calling another contract but there is no way the callee contract can change this address.

In the first three situations, the target address will be appeared in the runtime bytecode of the contract. For every proxy-based `DELEGATECALL`, we obtain the target address from the transaction's `to address`, and we obtain the caller's bytecode by invoking `eth_getCode` on the full node. If we find the target address in the bytecode, we mark it as a forwarder.

In the fourth case, we find where the target address is stored by the contract by decompiling the contract, with *Panoramix*[9], locating the line of code in the fallback function that makes the `DELEGATECALL`, and marking the storage slot for the target address. We parse the code and check if an assignment to that slot happens in any function in the contract—this is non-trivial and we refer the interested reader to the appendix of the full version of this paper [19] for the full details. 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 we mark the proxy as an upgradeable proxy contract.

Recall in the Universal Upgradeable Proxy Standard (UUPS) pattern, the logic contract implements a function to update the target address that is run in the proxy contract's context using `DELEGATECALL`. This is a subcase of the fourth case, where we check the logic contract instead of the proxy contract. If we determine the logic contract can assign values to the logic contract in any function, we tag it as UUPS.

In the fifth case, we rewind the transaction trace from the proxy-based `DELEGATECALL` and look for the target address being returned to the proxy contract in another action. If we find it being returned by a contract, we apply the methodology from the fourth case to this contract. If the target address is modifiable, we mark it as using the Beacon proxy upgradeability pattern. All contracts that remain after performing all of the checks above are marked as forwarders.

3.2 Results

Our measurements cover block number `10800000` to `12864595`, which corresponds to the time-period `Sep-05-2020` to `Jul-20-2021`, and are reported in Table 2. While we found 1.4M unique proxy contracts, many of these share a common implementation contract and are part of the same larger upgradable system. As one example, the NFT marketplace OpenSea [10] gives each user a unique proxy contract. After clustering contracts, we find 13K unique systems.

[9] `https://github.com/palkeo/panoramix`
[10] `https://opensea.io`

Proxy Contracts (Total)	1,427,215
Proxy Contracts (Filtered)	13,088
Regular Upgradeable Contracts	7,470
UUPS	403
Beacon	352

Table 2. Results of each `DELEGATECALL`-based upgrade pattern for the time-period Sep-05-2020 to Jul-20-2021 (2,064,595 blocks).

For the 8,225 upgradeable systems (regular, UUPS and beacon), we randomly sampled 150 contracts and manually verified they were upgradeable proxy contracts. We also sampled 150 contracts from the forwarders to verify they are not upgradeable, however we did find 2 false-negatives. Our model did not catch these contracts because a failure happened when decompiling them and our assignment checker detector in turn failed. Note that for UUPS contracts, the implementation contracts are much larger and harder to analyze than the proxy contract itself.

4 Finding the Admin

If a contract is upgradeable, someone must be permissioned to conduct upgrades. We call this agent the admin of the contract. In the simplest case, the admin is a single Ethereum account controlled by a private signing key, called an externally owned account (EOA). A breach of this key could lead to malicious updates, as in the case of the lending and yield farming DeFi service Bent Finance [1]. Bent Finance deployed a *Transparent Upgradeable Proxy* with an EOA admin that was breached (unconfirmed if via an external hack or insider attack). The EOA pushed an updated logic contract[11] which moved tokens valued at $12M USD into the attacker's account[12] and then upgraded the logic contract to a clean version to cover-up the attack. Based on *The State of DeFi Security 2021* [3] report by Certik,[13] "centralization risk" is the most common attack vector for hacks of DeFi projects.

Control over upgradeability typically falls into one of three categories:

1. **Externally owned Address (EOA):** One private key controls upgrades. It is highly centralized and one malicious admin or compromised private key could be catastrophic. It is also the fastest way to respond to incidents. An EOA may also pledge to delegate their actions to an off-chain consensus taken on any platform, such as verified users on *Discord* or *Snapshot*, however with no guarantee they will abide by it. In our measurements, we cannot distinguish this subtype as these are off-chain, social arrangements.
2. **Multi-Signature Wallet:** Admin privileges are assigned to a multi-signature wallet, requiring transactions signed by at least m of a pre-specified n EOAs.

[11] `https://etherscan.io/address/0xb45d6c0897721bb6ffa9451c2c80f99b24b573b9`
[12] 0xd23cfffa066f81c7640e3f0dc8bb2958f7686d1f
[13] `certik.com`

This distributes trust, and tolerates some corruption of EOAs or loss of keys. There is no guarantee different EOAs are operated by different entities and may be security theatre put on by a single controlling entity.

3. **On-Chain Governance Voting:** A system issues a governance token and circulates it amongst its stakeholders. Updates are decided through a decentralized voting scheme where the weight of the vote from an EOA (or contract address) is proportionate to how many tokens it owns. This system is potentially highly decentralized, but the degree depends on the distribution of tokens (*e.g.,* if a single entity controls a majority of tokens, it is effectively centralized). Voting introduces friction: (1) a time delay to every decision—some critical functionality might bypass the vote and use quicker mechanisms (*e.g.,* global shutdown in MakerDAO), and (2) on-chain network fees for each vote cast.

4.1 Methodology

We conduct our measurement on the 7,470 regular upgradeable contracts from Section 3. 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 suggests specific arbitrary slots for upgradeable proxy contracts to store the *admin address*.[14] We first check this specific storage slot using `eth_getStorageAt` on the full node. If it is non-zero, we mark what is stored as the admin address. For non-EIP-1967 proxies, we use a process that is very similar to how we found the storage slot of the target address in Section 3. 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. We extract the access control check and mark the address authorized to run this function as the admin address.

Finding the admin type. Having the admin address, we can check if the account is an EOA by invoking `eth_getCode` on the address from the full node: if it is empty, it is an EOA. Otherwise, it is a contract address. The most common multisig contract is Gnosis Safe.[15] We automatically mark the admin type as multi-sig if we detect Gnosis safe. We then switch the manual inspection to find other multi-signature wallets (*e.g.,* MultiSignatureWalletWithDailyLimit, *etc.*) and add them to the data set.

In some cases, the admin address is itself a proxy contract—a pattern known as an Admin Proxy. This adds another layer of indirection. We are reusing our methodology for identifying proxy contracts to exact the real admin account, and the proceed as above. Further details of the methodology and implementation are provided in the appendix of the full version of this paper [19].

[14] Storage slot 0xb53127684a568b3173ae13b9f8a6016e243e63b6e8ee1178d6a717850b5d6103

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

	EIP-1967		Non-EIP-1967			
Type	Regular Admins	Admin Proxy	Regular Admins	Admin Proxy	Arbitrary Slots	Fixed Address
EOA	900	1202	1313	92	2	49
Multisig	255	567	104	16	10	36
Governance/Other	53	462				160

Table 3. Results of each admin type in upgradeable contracts for the time-period Sep-05-2020 to Jul-20-2021 (2,064,595 blocks).

4.2 Results

Of 7470 proxies, 3558 are controlled by an EOA address, 988 are controlled by a known multi-signature wallet, and 2924 addresses are remaining. Table 3 breaks down each sub-category for these. Of the latter 2924 addresses, these are either decentralized governance or another unknown type. After manual inspection, we note some of the unknown contracts use undefined or new patterns for implementing multi-sig contracts; our model has false negatives in detecting multi-signatures. The results demonstrate significant centralization risk in upgradeability: 48% of systems could be upgraded with the breach of a single signing key, and an additional 13% by potentially a small number of signing keys.

5 Concluding Remarks

In our paper, we find that `DELEGATECALL`-based data separation is the most prominent upgrade pattern in Ethereum in recent years. Our evaluation framework gives some hint as to why this is the case. It avoids the need for a social upgrade, as in contract migration or the `CALL`-based pattern (without a proxy). `CREATE2`-based metamorphosis was recently made possible (with the introduction of `CREATE2`) and its use might grow over time, however it shares one major drawback with contract migration: the need to migrate the whole state from the old contract for each update, even if the update makes minor changes to the logic of the contract. Metamorphic contracts also run the risk of Ethereum removing the `SELFDESTRUCT` opcode they rely on. A drawback of `CALL`-based patterns is the heavy instrumentation each new contract needs before it can be deployed, whereas in a `DELEGATECALL`-based (along with migration and `CREATE2`-based) upgrade pattern, developers can simply deploy the new logic contract exactly as it is written. Putting these reasons together, `DELEGATECALL`-based pattern is an attractive option on balance.

The main take-away from studying upgradeability on Ethereum is that immutability, as a core property of blockchain, is oversold. Immutability has already been criticized for being dependent on consensus—both technical and social [21]—however the widespread use of upgradeability patterns further degrades immutability. Finally, as we show, the prominence of contracts that can be upgraded with a single private key (*i.e.,* externally owned account) calls into

question how decentralized our DApps (decentralized applications) really are. If the upgrade process is corrupted through a key theft or by a rogue insider, the whole logic of the contract can be changed to the attacker's benefit.

One recent application of our research was finding all contracts that implement the UUPS upgrade pattern, which become important when a vulnerability is discovered in one of the best-known libraries for implementing UUPS. We describe how we can find potentially vulnerable contracts in the appendix of the full version of this paper [19]. While others had found some contracts by looking for specific artifacts left by the UUPS library, we improved the state of the art by looking for the generic pattern of UUPS.

A final discussion point concerns Layer 2 (L2) solutions, such as optimistic rollups and zk-rollups [10]. For the readers that are already familiar with them, their central component is a bridge contract that let computations be performed off of Ethereum (layer 1) and have just the outputs validated on Ethereum. If the bridge contracts is upgradeable, the rules for accepting L2 state are also upgradeable which means every L2 contract is de facto upgradeable even if it does not implement an upgrade pattern. We saw Ethereum override the consensus of the network to revert the DAO hack, which was a rare and contentious event. If a similar attack happened on a L2, reverting would be much simpler and not require a hard fork: the L2 could simply update the bridge contract. For this reason, the consensus override upgrade pattern may be less rare in the future.

Acknowledgements. Our measurements were possible thanks to `https://archivenode.io/`. We thank Santiago Palladino (OpenZeppelin) and the reviewers for comments and discussions that helped to improve our paper. J. Clark acknowledges support for this research project from the National Sciences and Engineering Research Council (NSERC), Raymond Chabot Grant Thornton, and Catallaxy Industrial Research Chair in Blockchain Technologies and the AMF (Autorité des Marchés Financiers). J. Clark and M. Mannan acknowledge NSERC through Discovery Grants.

References

1. Bent update. Tech. rep., Bent Finance, `https://bentfi.medium.com/bent-update-12ae69a41dc6`
2. Contract upgrade anti-patterns. Tech. rep., Trail of Bits, `https://blog.trailofbits.com/2018/09/05/contract-upgrade-anti-patterns/`
3. The state of defi security 2021. Tech. rep., Certik Company, `https://blog.openzeppelin.com/the-state-of-smart-contract-upgrades/`
4. 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/`
5. Chen, J., Xia, X., Lo, D., Grundy, J.: Why do smart contracts self-destruct? investigating the selfdestruct function on ethereum. ACM Transactions on Software Engineering and Methodology (TOSEM) **31**(2), 1–37 (2021)

6. Chen, T., Li, X., Wang, Y., Chen, J., Li, Z., Luo, X., Au, M.H., Zhang, X.: An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In: International Conference on Information Security practice and experience. pp. 3–24. Springer (2017)
7. Dhillon, V., Metcalf, D., Hooper, M.: The DAO hacked. In: Blockchain Enabled Applications, pp. 67–78. Springer (2017)
8. Fröwis, M., Böhme, R.: Not all code are Create2 equal `https://informationsecurity.uibk.ac.at/pdfs/FB-Ethereum-Create2.pdf`
9. He, N., Wu, L., Wang, H., Guo, Y., Jiang, X.: Characterizing code clones in the ethereum smart contract ecosystem. In: International Conference on Financial Cryptography and Data Security. pp. 654–675. Springer (2020)
10. McCorry, P., Buckland, C., Yee, B., Song, D.: Sok: Validating bridges as a scaling solution for blockchains. Cryptology ePrint Archive (2021)
11. Murray, P., Welch, N., Messerman, J.: Minimal proxy contract. EIP-1167 (2018)
12. Ortner, M., Eskandari, S.: Smart contract sanctuary `https://github.com/tintinweb/smart-contract-sanctuary`
13. PALLADINO, S.: The state of smart contract upgrades `https://blog.openzeppelin.com/the-state-of-smart-contract-upgrades/`
14. Perez, D., Gudgeon, L.: Dissimilar redundancy in defi. arXiv preprint arXiv:2201.12563 (2022)
15. Perez, D., Livshits, B.: Broken metre: Attacking resource metering in evm. arXiv preprint arXiv:1909.07220 (2019)
16. Pinna, A., Ibba, S., Baralla, G., Tonelli, R., Marchesi, M.: A massive analysis of Ethereum smart contracts empirical study and code metrics. IEEE Access **7**, 78194–78213 (2019)
17. Reijsbergen, D., Sridhar, S., Monnot, B., Leonardos, S., Skoulakis, S., Piliouras, G.: Transaction fees on a honeymoon: Ethereum's eip-1559 one month later. In: 2021 IEEE International Conference on Blockchain (Blockchain). pp. 196–204. IEEE (2021)
18. Rodler, M., Li, W., Karame, G.O., Davi, L.: {EVMPatch}: Timely and automated patching of ethereum smart contracts. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 1289–1306 (2021)
19. Salehi, M., Clark, J., Mannan, M.: Not so immutable: Upgradeability of smart contracts on ethereum. Tech. rep., arXiv (2022)
20. Victor, F., Lüders, B.K.: Measuring ethereum-based erc20 token networks. In: International Conference on Financial Cryptography and Data Security. pp. 113–129. Springer (2019)
21. Walch, A.: The path of the blockchain lexicon (and the law). Rev. Banking & Fin. L. **36**, 713 (2016)