Proxy Hunting: Understanding and Characterizing Proxy-based Upgradeable Smart Contracts in Blockchains

William E Bodell III
Illinois Institute of Technology
wbodell@hawk.iit.edu

Sajad Meisami
Illinois Institute of Technology
smeisami@hawk.iit.edu

Yue Duan
Illinois Institute of Technology
yduan12@iit.edu

Abstract

Upgradeable smart contracts (USCs) have become a key trend in smart contract development, bringing flexibility to otherwise immutable code. However, they also introduce security concerns. On the one hand, they require extensive security knowledge to implement in a secure fashion. On the other hand, they provide new strategic weapons for malicious activities. Thus, it is crucial to fully understand them, especially their security implications in the real-world. To this end, we conduct a large-scale study to systematically reveal the status quo of USCs in the wild.

To achieve our goal, we develop a complete USC taxonomy to comprehensively characterize the unique behaviors of USCs and further develop USCHUNT, an automated USC analysis framework for supporting our study. Our study aims to answer three sets of essential research questions regarding USC importance, design patterns, and security issues. Our results show that USCs are of great importance to today's blockchain as they hold billions of USD worth of digital assets. Moreover, our study summarizes eleven unique design patterns of USCs, and discovers a total of 2,546 real-world USC-related security and safety issues in six major categories.

1 Introduction

Smart contracts [19] are programs deployed to blockchain networks that enable interactions between mutually distrusting parties without relying on a trusted third party. Due to the underlying blockchains, smart contracts are typically thought of as immutable once deployed - not even the creator of a smart contract can change its code. Therefore, smart contracts are widely considered trustworthy and have revolutionized or created many fast-growing businesses, including Decentralized Finance (DeFi) (e.g., Uniswap [68]) and NFT trading (e.g., OpenSea [51] and Looksrare [43]). The total market capitalization of smart contract-based assets is also booming and is expected to reach $770 million by 2028 [56]. While immutability is often considered necessary for the trustlessness and security of smart contracts [49], as is the case for the distributed ledgers they run on, it could be a drawback in many practical settings. For instance, like other programs, smart contracts may require updates for security (e.g., bug fixes) and non-security (e.g., new feature implementation) reasons. To address these issues without the need to migrate all activity to a new contract address, new patterns, namely **Upgradeable Smart Contracts** [75] (USCs), have become a trend of future smart contract development and are widely adopted by many companies, such as Compound Finance [13] and OpenSea [51].

Along with this trend, security problems are also emerging. On the one hand, implementing USCs in a secure fashion requires extensive security knowledge [76]. Although certain Ethereum Improvement Proposals (EIPs) [1–4] and some third-party libraries [16, 75] have been presented for implementing different upgradeable patterns, there is no de facto standard for developers to implement a USC correctly and efficiently. Incorrectly implemented USCs can lead to serious security loopholes. For example, in certain USC patterns [76], instead of using the standard *constructors*, developers should use self-defined *initializers* to initialize global variables so as to avoid *uninitialized variable vulnerabilities*. On the other hand, USCs also provide new strategic weapons for malicious activities. A recent 2020 report [5] has shown that attackers leveraged USCs to launch sophisticated attacks and keep evolving their attack schemes, resulting in huge financial loss. Clearly, as USCs are getting increasingly prevalent, they have become a new cybersecurity battleground for the security community to fight against attacks and vulnerabilities.

Despite the fact that many research efforts have been made on normal smart contracts to detect and alleviate security problems, such as detecting smart contract vulnerabilities [12, 14, 31, 32, 34, 36–38, 41, 44, 47, 48, 50, 61, 62, 65, 66, 70, 71, 73], mitigating security prob-

lems [33,57,58,77] and inspecting for fairness and safety issues [18, 39, 54, 64, 67], no comprehensive study has been conducted to help the community understand the status quo of USCs in the real-world, which is crucial to building practical defenses and mitigating the security risks brought in by these techniques.

In this paper, we report our systematic study on USCs in the real world which investigates a broad spectrum of USCs and characterizes them in terms of their security implications. Specifically, we seek to answer three sets of essential research questions.

(1) **Importance.** First, we would like to find out the importance of USCs for today's blockchain world. *How widely used are USCs in today's mainstream blockchains? How much USD worth of cryptocurrencies and tokens is currently held by USCs? How have these numbers changed over time?*

(2) **Unique Behaviors and Design Patterns.** Second, we dive into technical details and systematically reveal unreported unique behaviors and design patterns. *How can we characterize the uniqueness of USCs in the real world? How are USCs implemented? What are the unique behaviors and design patterns?*

(3) **Security and Safety Risks.** Third, we seek to study the security implications of USCs in the wild. *What are the security and safety issues associated with USCs? What is the possible impact of each issue?*

To answer these essential research questions, we need to effectively and automatically detect USCs in the wild and perform further behavioral and security analyses in a systematic way. To this end, we first develop a complete taxonomy that can comprehensively characterize USCs in terms of upgradeability, with syntactic and semantic features derived from the Solidity programming language specification [63]. Then, we report multiple types of security issues that are associated with USCs. Finally, we propose a novel system called USCHUNT, which is a static analysis framework for automatically detecting and analyzing USCs. To reliably detect USCs, capture their unique behaviors and perform security analysis, USCHUNT is equipped with cross-contract static analysis and inlined assembly modeling and relies only on intrinsic characteristics of USCs rather than heuristics. With it, we conduct our study on 861,657 smart contracts from 8 mainstream blockchains [6–8, 11, 21, 30, 52, 55] to answer the aforementioned important research questions. Here we highlight some interesting discoveries:

(1) Upgradeable smart contracts have become essential in today's blockchains as the numbers are growing exponentially and the total value held is over $3B.

(2) We discover 11 distinct USC design patterns and identify their unique strengths and weaknesses.

(3) We uncover and report a total of 2,546 real-world

USC-related security issues in six categories.[1]

Contributions. The contributions of the paper are summarized as follows:

- We develop a USC taxonomy, starting from the set of all elements that make up a smart contract, which systematically characterizes the unique behaviors of USCs at both syntactic and semantic levels.
- We design and implement a novel static analysis framework USCHUNT, which relies on intrinsic characteristics to detect and analyze USCs.
- We conduct a large-scale study on USCs with source code from 861,657 smart contracts on 8 mainstream blockchains.
- We open-source the implementation of our prototype USCHUNT and all study data/artifacts to facilitate future research[2].

2 Background and Related Work

2.1 Blockchains and Smart Contracts

A blockchain is a distributed ledger of transactions on a cryptographically secured peer-to-peer network, originally proposed in 2008 [46]. The ledger consists of a continuously growing list of ordered "blocks" that are tamper-proof and support non-repudiation. Each block contains transaction records and other state metadata, including the creation timestamp, the Merkle hash of the transactions, the hash of the previous block in the chain, and smart contract code and data.

Although early blockchain networks were used exclusively for recording cryptocurrency transactions, there were several attempts to build more advanced functionality on top of the Bitcoin blockchain. Ethereum [10] was the first blockchain to allow anyone to write smart contracts, where they can create their own arbitrary rules for ownership, transaction formats and state transition functions. Ever since, smart contracts have quickly become a powerful technique driving the transformation of multiple industries, such as NFT markets and DeFi.

2.2 Upgradeable Smart Contracts

Unlike typical software, smart contracts cannot be patched when a vulnerability is discovered, as their code is immutable. Also, while typical software can be taken offline when a bug is being patched, smart contracts run on decentralized networks that cannot be shut down for maintenance, and therefore cannot be easily paused.

---

[1]The disclosure is done by contacting USC creators via EthMail and their official communication channels when possible. Particularly, we manually search for the official communication channels of the contracts to report the issues. When we fail to find the official contact information, we use EthMail [15] service to send email alerts to inboxes accessible to the addresses of the contract creators.

[2]https://github.com/USCHunt-Anon/USCHunt

This makes performing smart contract upgrades nontrivial compared to traditional software updates.

However, for various reasons (e.g., bug fixes and new feature implementation), there has been a great demand for methods of making smart contracts upgradeable. An early strategy for upgradeability without the need for state migration was the so-called *data separation pattern*, which decoupled storage and logic by separating them into two contracts. In this pattern, the storage contract contains the state of the system, and only the logic contract can modify the state. A drawback of this pattern is the need to use a regular *call* from the logic to the storage contract to store and retrieve data, which is costly as each transaction requires a fee. The more common pattern, which virtually all upgradeable contracts have used in recent years, is the proxy-based upgradeability pattern. In this pattern, the proxy is immutable and stores the state of the system as well as a changeable pointer to the logic.
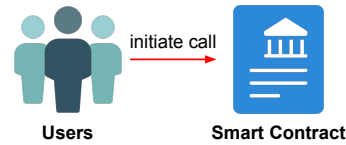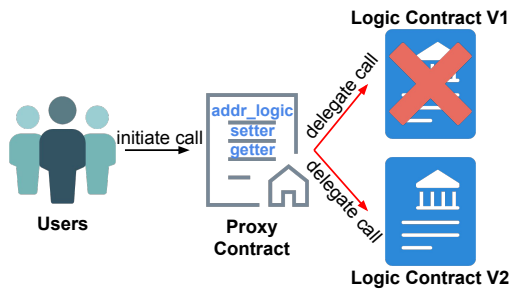

Figure 1: Normal Smart Contract.

Figure 2: Upgradeable Smart Contract.

Essentially, compared to a normal contract in Figure 1, where end users directly call the functions that contain actual business logic, a simple version of a proxy-based USC (e.g., auction) splits one contract into two: a proxy contract and a logic contract as shown in Figure 2. A proxy contract contains four important parts: the address of the current logic contract (denoted as $addr_{logic}$), the address retrieving code (denoted as *getter*), the address replacing function (denoted as *setter*), and a function for handling calls to functions in the logic (denoted as *fallback*). A proxy contract is responsible for interacting with users and delegating users' calls (e.g., place bid) to the logic contract that contains the actual business logic (e.g., the actual bid function). When upgrading, the developer deploys a new logic contract (Logic Contract V2) and calls *setter* to replace the old address in the proxy with the address of the new logic contract.

In general, USC proxy contracts use a special instruction (`delegatecall`) and a special function (*fallback*) to implement upgradeability. A `delegatecall` instruction will call an external function but execute the callee function in the caller's context. A *fallback* function, defined as `function()` or `fallback()` (since Solidity 0.6 [63]), is a special function that is executed on a call to the contract if none of the other function selectors (the first 4-bytes of the hash of the function signature) match that of the given function signature, or if no data was supplied at all. Using a `delegatecall` in the *fallback*, the proxy contract can forward users' calls to functions in logic contracts but execute them in its own context. Therefore, logic contracts do not need to store any users' data and hence, can be replaced easily. As shown in Figure 3, users call `Inc()` by providing its function selector (*0xccf19ee6*) along with the proxy's address. Since the proxy does not contain a function with the same function selector, the *fallback* function is executed, which uses `delegatecall` to call the actual `Inc()` in the logic contract and executes it in the proxy's context, resulting in an update of variable *x* in the proxy.
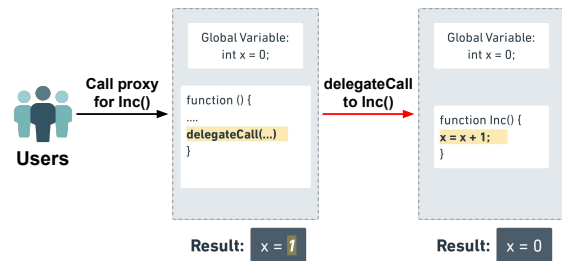

Figure 3: `delegatecall` Instruction

It is worth noting that there are other ways of implementing upgradeability, like the *data separation pattern* mentioned above. Another alternative is the so-called *metamorphic pattern*. It makes use of the special opcode `CREATE2`, which deploys a contract to a predetermined address, along with the `SELFDESTRUCT` opcode. This pattern destroys the original contract and then redeploys an upgraded version to the same address. It does not use a proxy, and therefore has a reduced overhead. However, it is generally frowned upon for several reasons and is extremely uncommon. As reported in the work [35], out of 32M contracts examined, only 41 smart contracts (0.00027%) have code updates using this pattern.

2.3 Security Analysis of Smart Contracts

Although little research has been done on USCs, a variety of techniques have been developed to perform analysis on regular smart contracts for detecting and mitigating security issues. These issues in smart contracts can be classified into two categories: low-level syntactic issues (e.g., reentrancy vulnerability [17]) and high-level

semantic issues (e.g., unfair business models).

Syntax-Level Security Analysis. Smart contracts suffer from multiple unique syntax-level security issues, such as transaction-ordering dependence, timestamp dependence, mishandled exceptions, unchecked low-level calls, and reentrancy vulnerabilities.

While static analysis is commonly used for vulnerability detection [36,61,62,65], some researchers adopt symbolic execution to explore the state-space of a contract and locate vulnerabilities [14, 31, 34, 41, 44, 48, 50, 66]. EthBMC [34], a bounded model checker, models EVM transactions as state transitions. teEther [40] generates constraints along a critical path that contains attacker-controlled instructions. Another line of research is leveraging fuzzing techniques for vulnerability detection [12,32,37,38,47,70,71,73]. Confuzzius [32] implements a hybrid fuzzer, and xFuzz [73] utilizes machine-learning to guide the fuzzing.

For bug mitigation, Sereum [57] modifies the EVM to defend against reentrancy attacks. EVMPatch [58] uses hard-coded templates to convert regular contracts into patched USC proxies. At the same time, Elysium [33] combines template-based and semantic-based patching and performs bytecode rewriting for a bug fix.

Semantic-Level Security Analysis. In comparison to syntax-level security issues, semantic-level security issues, such as unfair business models, are generally more complicated to detect as it requires extra semantic-level information. A series of research [9,18,39,54,60,64,67, 69] has been conducted to leverage formal verification to verify the correctness of certain properties that include semantic-level issues. Zeus [39] leverages abstract interpretation and software model checking. VetSC [18] uses natural language processing techniques to infer the semantics of smart contract functions, and leverages static analysis and model checking to model the business logic and further check the semantic-level safety issues. Sailfish [9] focuses on checking state inconsistency bugs.

3 USC Taxonomy

To conduct the study, we develop a taxonomy to differentiate and characterize USCs based on both syntactic and semantic features derived from our deep domain knowledge of USC and the Solidity language specification. Please note that the development of USC taxonomy is a one-time effort.

3.1 Syntactic Features
Our taxonomy development starts by dissecting every element within a smart contract and enumerating all possible values. According to the Solidity documentation [63]:

Definition 1. A smart contract can be defined as
$$S = (V, F, E, T, I), where: \qquad (1)$$

- V is a set containing all state variables in the smart contract;
- F is a set that includes external and internal functions, as well as *modifiers*;
- E is a set containing *events* and *errors*;
- T is a set of *struct* and *enum* types declared in the smart contract;
- I denotes the inheritance of the smart contract.

To ensure completeness, we carefully examine each element in this definition of a smart contract, to develop a set of low-level syntactic features which covers all possible aspects related to upgradeability, depicted in Table 4 in Appendix. We propose 24 different syntactic features that belong to six different feature sets, each of which is derived from one or more elements in a smart contract. The first column in the table shows the related elements from the smart contract definition $S = (V, F, E, T, I)$ and the others are the feature sets, syntactic features within each feature set and their possible values, respectively. For instance, the second feature set, $addr_{logic}$, which is one of the parameters of the `delegatecall` instruction, contains four distinct syntactic features, namely *Definition location*, *Type*, *Scope* and *Inheritance*. We also list all possible values for each. Based on the features and their possible values, it is easy to observe that the feature set is related to smart contract elements V, I, and T. Our feature sets contain comprehensive syntactic information of a USC since they cover every element in a smart contract except E, events and errors, which are irrelevant to upgradeability.

3.2 Semantic Features
Once all syntactic features and their possible values are identified, we then organically combine them to form a set of higher-level features – semantic features, each of which is composed of one or more lower-level syntactic features with specific values. These features contain not only the syntax-level information, but more importantly, they carry higher-level semantic information of USCs. We have identified 9 different semantic features as shown in Table 5 in Appendix. In the table, we list all the semantic features, their possible values, as well as the associated syntactic feature values.

Each semantic feature characterizes a unique behavior of a USC, and they fall into three major categories. First, *constant storage offset* and *storage layout coupling* features are related to how USCs define and store $addr_{logic}$. Then, the next four features, namely *simultaneous upgrades*, *partially upgradeable*, *scattered logic implementations*, and *transparent admin check*, denote how the upgradeability is implemented in a USC. Finally, the last three features are related to whether the upgradeability itself is modifiable or to the restrictions on upgradeability. For instance, if the semantic feature *removable upgradeability* is detected in a USC, it indicates that the

owner of the USC can choose to remove upgradeability voluntarily, without any user awareness.

4 USCHUNT Design and Implementation

To perform the large-scale study, we propose USCHUNT, an automated static analysis framework for detecting upgradeable proxy smart contracts, extracting both syntactic and semantic-level features for behavior analysis and vetting smart contracts for USC-related security analysis.

The reasons for detecting USC proxies are two-fold: 1) proxies have strong intrinsic characteristics that every USC must possess (shown in Section 2.2), hence, can be reliably detected; 2) once we have USC proxies, combined with some extra information (e.g., transactions), we can easily find their corresponding logic contracts.

4.1 Overview

USCHUNT performs static intra- and inter-procedural context-sensitive, flow-insensitive analyses of the Solidity program atop a state-of-the-art analysis framework Slither [61]. It transforms the smart contract code into a static single assignment IR named SlithIR for ease of implementation while preserving semantic information. As illustrated in Figure 4, as an analysis framework, USCHUNT implements an *analysis core* and enables three major analysis tools: *USC detector*, *feature extractor*, and *security analyzer*. The system takes two inputs, a smart contract source code and the developed USC taxonomy, and outputs the analysis results with respect to behavior and security. For a given smart contract, the *USC detector* can reliably tell whether it is a USC proxy. If yes, then USCHUNT will execute the *feature extractor* to extract syntactic and semantic features to characterize the USC, and further leverage the *security analyzer* to vet the smart contract and discover potential upgradeability-related security issues.

4.2 Analysis Core

As its name suggests, *analysis core* is an essential component in USCHUNT. It performs a variety of analyses on every component in a smart contract to collect useful information for USC detection as well as behavioral and security analysis. It contains four major parts, namely *fallback* analysis, *setter* analysis, *getter* analysis and other component analysis. The *fallback* analysis seeks to locate *fallback* function within a given smart contract and extracts useful information such as `dele gatecall` information and condition checks within the function. The *setter* analysis and *getter* analysis components are for identifying *setter* and *getter* for a given smart contract. We first extract the $addr_{logic}$ from the `delegatecall`, and perform inter-procedural backward data-flow analysis to trace back to its data origin. From there, we conduct forward data-flow analysis to find the data retrieving code (*getter*) and the data replacing function (*setter*). For other components, such as other state variables and external functions within the smart contract, we also perform analysis to collect information.

To achieve this, there exist two major technical challenges. First, our observations show that many USCs implement (at least part of) their *fallback* functions as inlined assembly code, which hinders static analysis. To tackle this challenge, we design a novel component in USCHUNT, the *assembly modeling component*, to automatically parse inlined assembly code, model it as different objects (e.g., variable objects, function call objects) and link them back to other contract code. Hence, we can have a complete view of the contract, including the inlined assembly. Second, major USC features, including $addr_{logic}$, *setter* and *getter* can be defined and stored in different smart contracts (e.g., proxy contract, logic contract, or other external contracts) in some complicated design patterns. As a result, we need to first find these related smart contracts and develop a cross-contract data-flow analysis to perform accurate analysis. To this end, we introduce a component, *explorer querying*, which queries the corresponding blockchain explorers [22–29] and collects all related smart contracts. Then, we instrument the original smart contract code to embed the related smart contracts into the same compilation unit and perform cross-contract analysis.

4.3 USC Proxy Detection

Given a smart contract, USCHUNT automatically detects if it is a USC proxy. Our work builds upon the static analyzer Slither [61], which itself comes with a native USC detector. However, it suffers some major shortcomings. Generally, Slither's upgradeability detection only checks for the keywords "upgradeable" and "proxy", and looks for `delegatecall` in the *fallback*, and it does not handle inlined assembly or cross-contract analysis.

In contrast, USCHUNT relies on the intrinsic characteristics of USC (i.e., delegation behavior in the *fallback* function, the *setter*, and the *getter*) to implement an effective detector. To capture these intrinsic characteristics, we follow Algorithm 1, with additional details provided in section 4.5. Given a smart contract, we leverage the *analysis core* in USCHUNT to locate the *fallback* function in which a `delegatecall` should reside (Ln.2-5). If this fails, the smart contract is confirmed not to be a USC proxy, and we return $false$. If it succeeds, we perform cross-contract data-flow analysis (CC_DFA) to trace the data origin of $addr_{logic}$. To do so, we follow an existing technique [72] to generate a cross-contract interprocedural control flow graph (XCFG) and perform data-flow analysis on top of it.

Once the origin of $addr_{logic}$ is extracted, we further call `CC_SetterAnalysis` and `CC_GetterAnalysis`
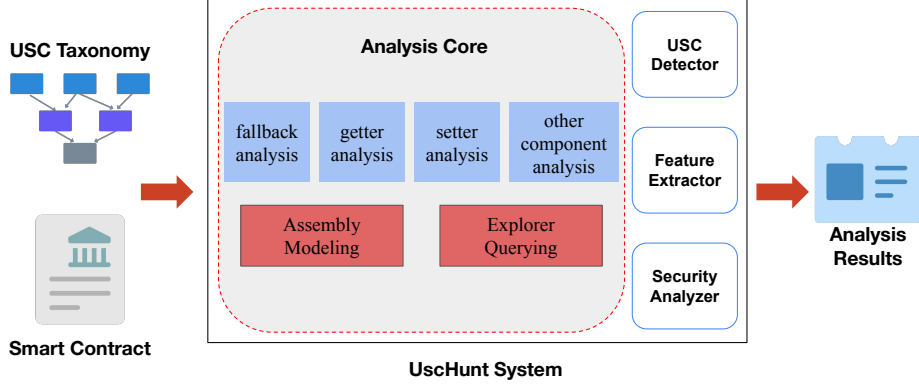
Figure 4: USCHUNT System Overview

(Ln.11-12) to identify the *setter* and *getter* by examining the def-use chain of $addr_{logic}$. If the *setter* exists, we can confirm that the given contract is indeed a USC proxy and return *true*. Locating the *getter* is often necessary for cross-contract analysis, i.e., when the *setter* is located in another contract, the *getter* will typically contain a call to this external contract.

Algorithm 1 USC Proxy Detection

1: **procedure** DETECTUSC(*sc*)
2:     $fb \leftarrow$ FALLBACKANALYSIS(*sc*)
3:     $asm \leftarrow$ MODELINLINEDASSEMBLY(*fb*)
4:     $fb \leftarrow fb \cup asm$
5:     $dc \leftarrow$ EXTRACTDELEGATECALL(*fb*)
6:     **if** *fb* is NULL OR *dc* is NULL **then**
7:         return false
8:     **end if**
9:     $EI \leftarrow$ QUERYEXPLORER(*sc*)
10:     $addr_{logic}Orig \leftarrow$ CC_DFA(*dc.tgt*,*EI*)
11:     $setter \leftarrow$ CC_SETTERANALYSIS(*addr_{logic}Orig*,*EI*)
12:     $getter \leftarrow$ CC_GETTERANALYSIS(*addr_{logic}Orig*,*EI*)
13:     **if** *setter* is NULL OR *getter* is NULL **then**
14:         return false
15:     **else**
16:         return true
17:     **end if**
18: **end procedure**

4.4 Behavioral & Security Analyses

Besides USC detection, our system also performs behavioral and security analyses on a given USC. The *feature extractor* component in USCHUNT is responsible for extracting syntactic and semantic features from the given USC, with the help of the USC taxonomy. To achieve this, USCHUNT relies on its core capabilities and performs a series of cross-contract static analyses including control- and data-flow analyses, and taint tracking on the given USC. Our system first scans over the contract to extract syntactic features and their values. Then, based on the mapping relationship between semantic and syntactic features shown in Table 5, USCHUNT identifies all the semantic features that are related to the given USC, and further performs static analysis to obtain their values. For instance, to extract the syntactic feature *Timestamp-related revert* in the *setter* feature set as shown in Table 4, our feature extractor first locates all `revert()` statements within the *setter* function. Then, it performs control-dependency analysis to collect all the conditional checks, on which the `revert()`s have direct control dependency. Furthermore, we run a backward data-flow analysis on the conditions to see if they are timing-related (i.e., comparison with time-related data such as *now*). If so, the feature *Timestamp-related revert* is identified.

The security analyzer seeks to automatically detect our discovered security issues (more details in Section 6.3). We leverage the analysis capabilities in USCHUNT to extract features that are related to security issues, and also synthesize the feature extraction results with a built-in upgradeability checker tool in Slither [61], which comes with methods for detecting storage layout clashes and function selector collisions. Beyond these results, USCHUNT also detects security issues that Slither's tool does not detect, such as missing or insufficient compatibility checks in the *setter*, while other issues are directly related to semantic features which our tool extracts, such as removable or time-delayed upgradeability.

4.5 Implementation Details

Here we elaborate on some important implementation details of USCHUNT.

Cross-Contract Data-flow Analysis. Our cross-contract data-flow analysis is built upon a cross-contract interprocedural control flow graph (XCFG) proposed in [72]. Once we extract `dc.tgt` used in the `delegatecall`, we then match it to either a *StateVariable* or *LocalVariable*, the latter can either be in the body of the current function or passed in as a parameter. If a *StateVariable* is found, then we've already found the $addr_{logic}$. In case a *LocalVariable* is located, we further perform backward interprocedural data-flow analysis on the local variable to eventually extract the corresponding state variable which is the data origin of the local variable `dc.tgt` used in the `delegatecall`. If at any point we come across an `sload(slot)`, we return the constant storage slot, as there is no other StateVariable.

Fallback Analysis. For each node in the fallback function's CFG, we first look for `delegatecall` in the Slither IR, and then search the assembly, which may be modeled as an *ASSEMBLY* node or as *EXPRESSION* nodes. For an *EXPRESSION* node, we simply extract the delegate variable. For an *ASSEMBLY* node, we need to model the assembly first and return the modeled variables.

Inlined Assembly Modeling. We modify Slither's solc parsing for functions, so that for Solidity versions $\geq$ 0.6.0, the *ASSEMBLY* node would contain the Yul (an intermediate representation for EVM assembly) AST. Then we parse it to find the `delegatecall` and extract the target argument `dc.tgt`. For earlier versions of Solidity where the assembly code is only a string, we modify the function's CFG based on a line-by-line interpretation of the code string and match the variable names to the original variable data structures.

`delegatecall` ***Target Extraction.*** If a `delegatecall` is found in a non-assembly code region, or in an assembly region modeled as an *EXPRESSION* node (for Solidity versions $\geq$ 0.6.0), we parse the Slither IR to extract the source variable directly. But when the function is found in an *ASSEMBLY* node, we model the inlined assembly to extract the name of the `delegatecall` target variable. Once we have the name of the target variable, we use `CC_DFA` to trace this local variable to its source.

Getter Analysis. `CC_GetterAnalysis()` iterates over all functions in the given contract, looking for the *getter* for a particular variable, skipping the *fallback*, *receive*, *constructor* and *initializer* functions. For each function, if the variable is not declared in the function's signature, it searches the function's CFG for either a *RETURN* node which returns the variable in question, or an *ASSEMBLY* node which uses `sload` to read from the implementation storage slot, if one was found during `CC_DFA`. The method halts and returns as soon as it finds the *getter*.

Setter Analysis. Similarly, *CC_SetterAnalysis*, iterates over all functions in the given contract, looking for the *setter* for a particular variable, skipping the *fallback*, *receive*, *constructor* and *initializer* functions as well as internal functions. It searches each function's CFG for either an *ASSEMBLY* node which uses `sstore` to write to the storage slot if one is given or a node containing an expression assigning a value to the $addr_{logic}$ variable. The method halts and returns as soon as it finds the *setter*.

4.6 Evaluation of USCHUNT

Next we evaluate the effectiveness and efficiency of USCHUNT with respect to USC detection. We use the contracts from Ethereum mainnet, which the original Slither identified as upgradeable proxies, to form a dataset that contains contracts with varying complexity.

We remove those which are not real proxies, and add some others ourselves, bringing the dataset to 994 contracts. Through manual investigation, we can confirm that our dataset contains 825 USCs and 169 non-USCs, and we use the information as ground truth. According to our inspection, these 169 were false positives in Slither's original USC detection, while four of the 25 contracts we manually added to this dataset were false negatives.

4.6.1 Effectiveness

We run USCHUNT on our dataset and report the false positive rate (FPR) and false negative rate (FNR). Our evaluation results show that our tool is able to correctly identify 812 confirmed USCs and misses only 13, indicating a rate of FNR of 1.30%. Moreover, the tool does not mislabel any non-USC, therefore, has no false positive. This is is because USCHUNT is designed to accurately find all of the intrinsic characteristics of USCs, such as *setter* and *getter*, and then detect USCs.

We further investigate all 13 false negative cases to understand why they occur. All of them are related to cases where $addr_{logic}$, *setter* or *getter* are defined in some external contracts. However, our tool fails to find the contracts via transaction analysis and explorer querying, and hence mislabels these cases as non-USC. In a nutshell, USCHUNT can effectively detect USCs with very high accuracy.

4.6.2 Efficiency

On average, USCHUNT takes 0.764696 seconds to analyze a contract. As illustrated in Figure 5, it can process more than 83.6% of contracts (831 out of 994 contracts) in less than one second. Therefore, we can conclude that USCHUNT works efficiently enough to perform large-scale smart contract analysis.
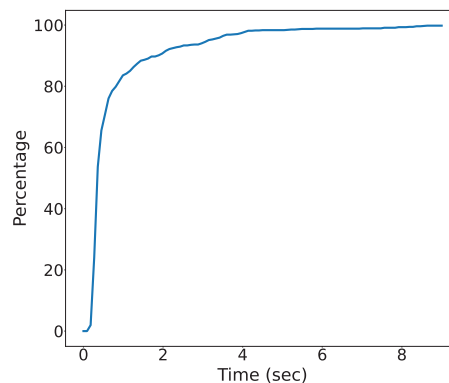

Figure 5: CDF diagram of USCHUNT execution time

5 Study Methodology

To answer the essential research questions brought up in Section 1, our study follows a well-defined methodology. This section elaborates on the methodology that we have systematically identified and itemized to facilitate the answers to each and every question.

To conduct the study, we use a large smart contract dataset [53], which contains 800K+ contracts from 8 mainstream blockchains (Arbitrum, Avalanche, Binance Smart Chain, Celo, Ethereum, Fantom, Optimism and Polygon) with deploy dates ranging from 2017 to 2022.

5.1 Methodology

For each set of research questions, we elaborate our methodology by listing four important aspects: 1) challenges, 2) solutions, 3) detailed analysis, and 4) limitations. The challenges and solutions sections are to list all the technical challenges to be addressed during the study as well as our proposed solutions. The analysis section describes the proposed analysis to be performed to explore the answer, while the limitations section is elaborated to discuss the possible limitations of our analysis.

5.1.1 (RQ1) Popularity and Value Over Time

What is the popularity of upgradeable proxy contracts? How much total value is held by upgradeable proxies? How have these changed over time?

Challenges and Solutions. There exist two major technical challenges for answering RQ1. First is how to effectively identify USCs in our large dataset, and get the total value held. The second challenge lies in how to extract the changes in these numbers over time. To resolve the first challenge, we leverage the USC detector in UscHunt to detect the existence of USCs in our dataset and add up the numbers. Further, to obtain the total value held, we automatically scrape the blockchain explorers to extract the current cryptocurrencies and tokens held by each USC address and calculate the total value by their exchange rate with USD.

Analysis. Having used UscHunt to extract the subset of the smart contracts in our dataset which are USCs, we first run a simple script to count the total number of contracts in our dataset, and the number of those which were flagged as USC proxies, and compute the percentages for each chain. We then compile a list of all the addresses of these proxies, and then run a handful of scripts to collect the deploy date and total value data for each chain.

Limitations. A major limitation is the missing USCs since UscHunt is based on source code static analysis, therefore, cannot analyze smart contracts with only bytecode. Moreover, UscHunt may also impose certain false negatives.

5.1.2 (RQ2) USC Behaviors and Design Patterns

How can we characterize the uniqueness of USCs in the real world? How are USCs implemented? What are the unique behaviors and design patterns?

Challenges and Solutions. The biggest challenge for answering RQ2 is how to define USC behaviors and design patterns. To this end, we consider both syntactic and semantic features of a USC to uniquely characterize its be-

haviors. We first develop a list of syntactic features from Solidity specification [63], and make sure our features cover all major components in a smart contract. After that, we enumerate all possible values of these features and organically combine them to form a list of semantic features that can narrate high-level behavioral information of USCs, as our semantic features. Eventually, we further enumerate all possible values of semantic features and leverage real-world samples to summarize the unique design patterns of USCs.

Analysis. By running the *feature extractor* in UscHunt on the USC dataset formed in RQ1, we perform automatic analysis on USCs and extract all features and their values based on the taxonomy. Once we have the features and their values, we then distill higher-level design pattern information by combining multiple features and their values together and come up with a list of unique design patterns of USCs.

Limitations. Although the syntactic and semantic features from the USC taxonomy are systematic since they originate from the language specification, our design pattern list is completely developed from the samples in our dataset. Therefore, the list may not be exhaustive.

5.1.3 (RQ3) Security Risks

What are the security and safety issues associated with USCs? What is the possible impact of each issue?

Challenges and Solutions. There are several challenges. First, we need to identify a set of upgradeability-related security issues. Second, it can be challenging to automatically detect these issues as there exists no existing tool available. To handle the first challenge, we systematically examine every aspect of USC and summarize five different security issues with respect to upgradeability. For the second challenge, we develop a security analyzer that can leverage the *analysis core* in UscHunt to automatically perform security analysis on USCs.

Analysis. To conduct the study, we feed all the discovered USCs from RQ1 into the *security analyzer* in UscHunt to discover potential security issues. Then, based on different security issues, we investigate the discovered issues and try to find the real-world impact that these issues could possibly bring to all parties including end users, developers, and other contracts.

Limitations. Compilation errors may occur when instrumenting the proxy source code with the logic contract, thus, preventing us from analyzing them for security issues. Also, some security issues–such as storage layout clashes–are so well documented that we rarely find them manifested as bugs in real-world contracts. Others–such as function selector collisions–are prevented by design in most USC patterns and are particularly rare even in non-standard USC designs.

6 Findings

Below, we present our findings with regard to the research questions described in the previous section.

6.1 RQ1: Importance

Table 1 shows the total number of USC proxies that UscHunt detected, and the percentage of the contracts in our dataset for each chain.

Table 1: Percentage of USCs on Each Blockchain

Chain	Total Count	USC Proxy Count	USC Proxy Percent
Ethereum	482,889	5,384	1.11%
Arbitrum	4,684	189	4.04%
Avalanche	29,759	282	0.95%
BSC	261,068	1,507	0.58%
Celo	917	56	6.11%
Fantom	16,893	218	1.29%
Optimism	960	60	6.25%
Polygon	64,487	1,119	1.74%
Total	**861,657**	**8,815**	**1.02%**

To answer the remaining questions in (RQ1), we turn to the corresponding blockchain explorers [22–29]. Table 2 shows the total value held by USC proxies on each chain, including their native coins (e.g., ETH for Ethereum, BNB for Binance Smart Chain) as well as ERC-20 tokens, at USD prices as of Sept. 30, 2022. We note that the value of ERC-20 token holdings is typically orders of magnitude greater than the native coin value. In addition to this value held in the proxies themselves, we manually locate an additional $443.5M held in related contracts accessible to the proxies on Ethereum, and an additional $89M on BSC.

Table 2: Total value held in USCs

Chain	Native Value	Token Value	Total Value
Ethereum	$68.1M	$2.6B	$2.7B
Arbitrum	$16.1K	$2.5M	$2.5M
Avalanche	$6.9K	$4.7M	$4.7M
BSC	$177K	$12.5M	$12.7M
Celo	$322.6M	$106.2M	$428.8M
Fantom	$304.6K	$3M	$3.3M
Optimism	$0	$425.5K	$425.5K
Polygon	$43.3K	$1.4M	$1.4M

Figure 6 plots the change in the number of USC proxies deployed over time. The blue line indicates the total number in our dataset across all mainnet chains, which is the sum of the eight lines below it. The first USC contracts deployed from 2018 to mid-2020 coincide approximately with the introduction of the unstructured storage pattern in OpenZeppelin's earliest upgradeable contracts, as well as the initial submissions of the EIP-1822 [2] and EIP-1967 [3] standards.

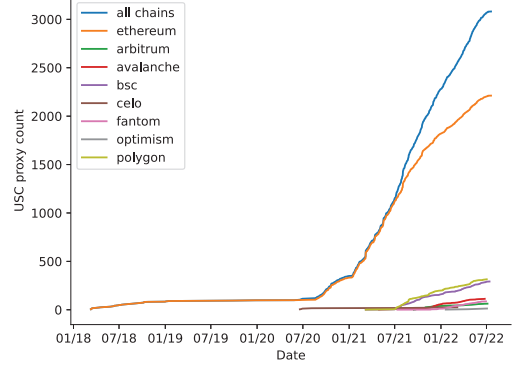Figure 8 in the appendix answers the additional question, how often are these USCs upgraded? We run

Figure 6: Number of USCs Deployed By Date

UscHunt, which identifies the $addr_{logic}$ $setter$ function, and use this knowledge to search the transaction history of each USC.

By answering RQ1, we can see several notable inflection points: first, USCs have been increasingly popular in the past two years. Especially, around July 2020 we note a substantial increase in the rate of USC proxy deployment, which we attribute to the release of OpenZeppelin's Upgrades plugin, which makes the USC proxy easier to implement. Second, USCs currently are holding billions of USD worth of cryptocurrencies and tokens. Consequently, we can conclude that USCs are really important in today's blockchain world.

6.2 RQ2: Patterns and Behaviors

By detecting and combining semantic and syntactic features from real-world USCs, we discover unique design patterns in the wild. In total, we have discovered 11 USC design patterns. Table 6 in Appendix presents the discovered USC design patterns, each of which is defined by the combination of several semantic and syntactic features. Figure 7 further displays the numbers of each USC proxy pattern detected on each chain in our dataset.
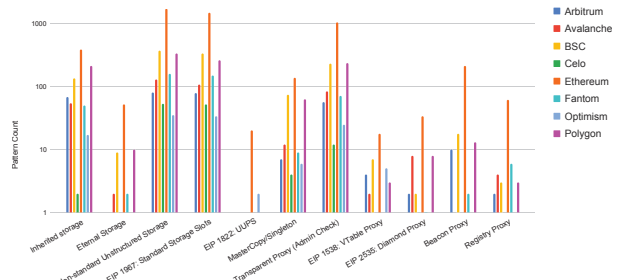

Figure 7: Number of USC Design Patterns Detected

Storage Patterns. Different patterns have their own motivations and address different upgradeability-related concerns. The first five patterns in Table 6 deal primarily with storage layout compatibility. Inherited Storage dictates that both proxy and logic should inherit the

same base storage contract, with any additional storage variables being appended to the prior storage contract via inheritance. Eternal Storage uses mappings for each type of Solidity variable, as values stored in mappings do not take up slots at the beginning of the storage space [20]. However, we find that Eternal Storage is almost always combined with Inherited Storage, with the $addr_{logic}$ typically stored as a state variable rather than in the address mapping. Unstructured Storage, with its three sub-patterns, uses a specific storage slot at an arbitrarily high offset for each variable that must be accessed by the proxy, such that the logic contract requires no knowledge of the proxy's storage layout. The earliest Unstructured Storage proxies, introduced by OpenZeppelin, used the keccak256 hash of `org.zeppelinos.proxy.implementation` as the implementation storage slot, and other projects replace this with their own strings. EIP-1967 introduces a set of standardized storage slots, using `bytes32(uint256(keccak256('eip1967.proxy.implementation')) - 1)` for the implementation. EIP-1822, a.k.a, the Universal Upgradeable Proxy Standard (UUPS), predates EIP-1967 and uses `keccak256('PROXIABLE')`.

Function Selector Patterns. Three of the patterns are motivated by the need to avoid function selector collisions between the proxy and logic contracts. EIP-1822 is a variant of Unstructured Storage, in which no functions other than the constructor and *fallback* are declared in the proxy contract, whereas the *setter* is located in the logic, and in place of a *getter*, the implementation storage slot is hard-coded in the *fallback*. The Mastercopy/Singleton pattern takes a similar approach, including only a *fallback* and constructor in the proxy code, though it stores its implementation in a state variable located at slot 0, thus requiring that the logic contract always declare the same variable in the first storage slot, so there is a strong coupling between the two. The Transparent Proxy pattern takes a different approach to avoiding function selector collisions: it requires that the proxy also stores an admin address, and only allows calls to the proxy's external functions by the admin while disallowing calls from the admin being handled by the *fallback*. Thus, if a function in the proxy and another in the logic both share the same function selector, the admin account can only ever execute the former, while calls from any other address will always be routed to the latter. The Transparent Proxy pattern and EIP-1822 (UUPS) frequently use the standard storage slots of EIP-1967.

Atomic Implementation Patterns. The following two columns in Table 6, EIP-1538 [1] and EIP-2535 [4], make use of mappings with 4-byte function selectors as keys, such that each function can be associated with a separate implementation contract, effectively skirting the limitation on smart contract size enforced by the EVM while allowing atomic upgrades limited to individual functions. EIP-2535, a.k.a, Diamond or Multi-Facet Proxy standard, introduces a solution to prevent function selector collisions and a novel approach to storage layout, which is similar to Unstructured Storage but packs several variables and mappings into a *struct*, and then stores it at a specific storage slot.

Registry Patterns. While the preceding patterns typically only allow one proxy to be upgraded at a time, the last two patterns (Beacon and Registry) allow any number of proxies to be upgraded at once by having them retrieve their implementation address from an external contract, which contains the upgrade functionality rather than the proxy itself. These patterns are characterized by the presence of a call to an external contract address other than the `delegatecall` target in either the *getter* or the *fallback*, which should return an address value that is stored in the external contract. In our detection, we differentiate between Registry and Beacon Contracts based on the presence or absence of arguments passed into the external call, as well as how the implementation address(es) are stored in the contract. By definition, a Beacon contract should store only one implementation address, obviating the need for arguments in the *getter*, whereas a Registry may store many implementations, typically in a mapping, and returns the one associated with a given key. The Beacon Proxy pattern is very often combined with EIP-1967, using the hash of `eip1967.proxy.beacon` for the Beacon's address.

6.3 RQ3: Security and Safety Issues

We follow the study methodology to answer RQ3, and discover 2,546 real-world USC-related security and safety issues in six major categories, as shown in Table 7 in Appendix. Note that these issues may not be directly exploitable, but could become serious bugs if the contract admin is not careful or less experienced. Our findings can be broken down into two categories: issues related to the implementation of upgradeability itself, and policy issues related to how or when upgrades can occur.

6.3.1 Implementation Issues

Storage layout clashes. Storage layout clashes can manifest in two ways: between the proxy and logic contract, or between two logic versions. From an EVM storage perspective, the first state variable assigns to slot zero, the next to slot one, and so on. If a contract uses inheritance, state variables in the inherited contracts are stored, according to these rules in the order of inheritance, before those declared in the contract itself. Storage clashes between proxy and logic occur if both declare state variables in a typical way, and the logic contract can accidentally overwrite the values of those in the proxy since

`delegatecall` executes in the caller's context. A storage layout clash happens between two logic versions if a USC upgrades to a new logic with the variable order changed. The proxy's storage layout will not reflect these changes and values may be overwritten.

Listing 1 presents a simplified version of the former, derived from a real-world DeFi Synthetix. The storage layout clash is in the first lines of each contract: the Proxy contract stores its $addr_{logic}$ as a Logic contract-type state variable in the first storage slot (Ln.2), while the logic contract declares another variable (`otherAddr`) in the same position (Ln.13). The second example (Listing 2), also derived from a real-world USC DexProxy, exhibits both sorts of storage layout clashes, between proxy and logic as well as between two logic versions, causing the proxy to become impossible to upgrade.

```
1  contract Proxy {
2    Logic public target;
3    function setTarget(Logic _target) {
4      target = _target;
5    }
6    function() external payable {
7      ...
8      delegatecall(gas,sload(target), ...)
9    }
10 }
11
12 contract Logic {
13   address public otherAddr;
14   function setOtherAddr(address _other) {
15     otherAddr = _other;
16   }
17 }
```

Listing 1: Synthetix Proxy and Logic Contracts

To implement the Inherited Storage pattern correctly, it is essential that the different versions of logic contracts follow the same order of inheritance. Even the earliest logic contract for this proxy was implemented incorrectly, as its declaration read `contract Dex is Ownable, DexStorage`, such that the two slots occupied by the variables in ProxyBaseStorage, inherited first by the proxy, were replaced by the single variable declared in Ownable, inherited first by the logic. This had the effect of shifting the storage locations of all variables declared in DexStorage up by a slot.

The problem happens when the developer upgrades to a new logic implementation, which adds several new variables to the DexStorage contract, as shown in Listing 3. Note the new variable (ln. 2) added before the two pre-existing ones, and more importantly, the new boolean variable `openForSale` (ln. 6) now clashes with the proxy's `_owner` address (ln. 10) in the previous listing. Thus, when the owner proceeded to call the new `setSaleClose()` function, they overwrote the last digit of the `_owner` address with a zero. Because only the owner is authorized to upgrade the contract, thus made upgrading effectively impossible.

Function Selector Collisions. A function selector, which is the first 4-bytes of the hashed function signature, is needed for making a function call. Function selector collisions occur when two functions happen to have the same function selector. It becomes a security issue when it happens between a function in the logic and one in the proxy, where attempts to call the logic function can unintentionally be handled by the proxy function instead. If the proxy function in question happens to be *setter*, it sets *addr_logic* to an undetermined address when trying to call a different function provided by the logic contract.

```
1  contract ProxyBaseStorage {
2    mapping(bytes4 => address) delegates;
3    bytes[] public funcSignatures;
4  }
5  contract DexStorage {
6    address payable public platform;
7    uint256 internal platformPerecentage;
8  }
9  contract Ownable {
10   address internal _owner;
11 }
12 contract DexProxy is ProxyBaseStorage,
       DexStorage, IERC1538, Ownable {
13   function() external payable {
14     address deleg = delegates[msg.sig];
15     assembly {
16       ...
17       let res:= delegatecall(gas,deleg,...)
18       ...
19     }
20   }
21 }
```

Listing 2: DexProxy and its inherited storage

```
1  contract DexStorage {
2    address nft;
3    address public platform;
4    uint256 internal perecentage;
5    mapping (address => bool) collections;
6    bool openForSale;
7  }
8  contract Dex is Ownable, DexStorage {
9    ...
10   function setSaleClose() public {
11     require(admin==msg.sender,"not admin");
12     openForSale = false;
13   }
14   ...
15 }
```

Listing 3: Upgraded logic and storage for DexProxy

Although we have not detected any real-world examples of function selector collisions in our dataset, we can easily form a possible scenario. Take the function `setTarget(_target)` from Listing 1 as an example. It has the 4-byte selector `0x776d1a01`, and it is the same as for the function signature `unvest(uint256, uint256, uint256, uint256, uint256, bool)`. If this function were added in a future upgrade to the logic contract, it would not be callable, as all attempts to call the new function would be handled by `setTarget(_target)`, which is the *setter* in the proxy. If the contract's owner tries to call the `unvest` function they could end up overwriting *addr_logic* with the value of the first argument.

Insufficient Compatibility Checks. When performing an upgrade, certain compatibility checks need to be en-

Table 3: Detected Compatibility Checks by Class

Chain	Checks contract call result	Checks address is a contract	Checks address is not zero	Checks new not same as the old	Missing checks
Ethereum	28	1337	79	219	694
Arbitrum	0	80	9	2	76
Avalanche	6	78	5	17	99
BSC	10	295	36	41	260
Celo	0	11	6	0	19
Fantom	0	65	14	1	151
Optimism	0	23	1	1	31
Polygon	5	242	37	65	281
Total	49	2131	187	346	1611

forced in the *setter* to ensure that the new target is compatible with the USC, otherwise, an upgrade may replace the existing target with a pointer to an arbitrary address. In some cases, this may cause storage layout clashes if the new target contract uses an incompatible storage layout. In the worst case, if the *setter* is meant to be in the logic contract but an upgrade sets the target to some other address, such an upgrade can make the USC completely unusable. Note that the compatibility checks we discuss do not consider the business logic or semantic structure of the contract, only matters related to preserving upgradeability and overall functionality.

Through our study, we detect a substantial number of USCs which are either missing a compatibility check or have some form of checks that are insufficient to prevent erroneous upgrades. Based on the compatibility checks extracted during our analysis, we categorize the checks into four types and provide an example of each below. Table 3 gives the number of each class of compatibility check detected on each mainnet.

(1) **Checks contract call result.** In the best case, the *setter* should attempt to call a function that is expected to be in the new logic contract, and verify that it returns the correct value, as EIP-1822 does: `require(bytes32(PROXIABLE_MEM_SLOT) == Proxiable(newAddress).proxiableUUID())`

(2) **Checks address is a contract.** It may be sufficient to check that the new address is indeed a contract: `require(extcodesize(newLogicAddr) > 0)`

(3) **Checks address is not zero.** A common yet insufficient check ensures the new address is not zero: `require(_implementation != address(0))`

(4) **Checks new address is not the same as old.** The least sufficient check, no bearing on compatibility: `require(currentImpl != _newImpl)`

In addition to these common classes of compatibility checks, we also discover another related issue. While Listing 1 appears to check the type of the new target contract, due to its function signature `setTarget(Logic _target)` using a specific contract-type argument, internally no type checking is ever performed. Once the contract is deployed, the EVM treats such variables just as it would an ordinary address, so this contract actually has no compatibility check. In total, we detect a whopping 2,243 real-world samples with this security issue.

6.3.2 Policy Issues

Upgradeability Can Be Removed Accidentally. If the *setter* exists outside of the proxy contract (i.e., either in the logic or an external contract), the upgradeability could be accidentally eliminated permanently. While this may also occur intentionally, it is a potential safety issue for USCs using a pattern in which the setter is located in either the logic or an external contract.

Considering how prevalent the insufficient compatibility checks are, this issue certainly is worth reporting. As shown in Table 7, we detect 213 such cases. It simply reflects the total number of contracts in our dataset that exhibit the *removable upgradeability* semantic feature, as this safety issue is directly related to that feature. It also comes to our attention that while some EIPs, such as EIP-1822, warn that the *setter* must be present in the logic contract to retain upgradeability, others (e.g., EIP-2535) take a different stance, arguing that the ability to make their code immutable is a feature, not a bug.

```
1  contract EnclavesDEXProxy {
2    address public proposedImpl;
3    uint256 public proposedTimestamp;
4
5    function propose(address _proposed) {
6      ...
7      proposedImpl = _proposed;
8      proposedTimestamp = now + 2 weeks;
9    }
10   function upgrade() {
11     ...
12     require(proposedTimestamp < now);
13     impl = proposedImpl;
14   }
15 }
```
Listing 4: Example of a USC proxy with a time-delay.

Vulnerabilities Can't Be Patched On-time. This specific security issue is a direct result of the *time-delayed upgrades* high-level semantic feature presented in Section 3. As shown in Listing 4, it is defined by a timestamp variable declared in the proxy contract, which is checked in the *setter* function before performing the upgrade operation. In this case, if a vulnerability is discovered in the USC and needs to be patched, a malicious user has this time window before the next upgrade to exploit it.

This issue has caused real-world attacks. In Sep 2021, an upgrade was performed on Compound [13], which is a popular DeFi protocol with a total value locked (TVL) of $10.2 billion [42]. The upgrade introduced a logical bug in the protocol's Comptroller contract [74]. However, since upgrades require a seven-day time delay, developers could not fix it immediately, causing $100 million loss in tokens. Therefore, developers should always either include an emergency upgrade/rollback function, or a means to pause the system until a fix goes into effect.

6.3.3 False Positives and False Negatives

We manually inspect all contracts that are flagged as having security issues and confirm that there exists no FP.

Similarly, we rule out all FNs other than those stemming from a failure to identify a contract as an upgradeable proxy, such as cases where we are unable to obtain the source code of the logic contract for an apparent UUPS proxy, and thus unable to locate the *setter*. For this reason, the false negative rate w.r.t. security issues is the same as what is reported in section 4.5.1.

7 Discussion

In the following section we present some final observations regarding USCs, based on findings from the study.

7.1 Trade-Offs

Choosing the Right Pattern. Through the study, we observe a number of trade-offs being made by USC developers in choosing between patterns. For instance, when deciding how to address storage layout concerns, each pattern has its pros and cons. Inherited Storage is certainly the most intuitive, and it allows the proxy to have access to as many state variables as a developer may feel necessary, yet it can easily be implemented incorrectly with dire consequences, as can Eternal Storage. Unstructured Storage has the drawback of requiring blocks of assembly code to read and write storage variables, but the standards in place have been well-tested.

To Upgrade or Not to Upgrade? Can we trust the developers authorized to perform a smart contract upgrade? This question naturally leads to perhaps the biggest trade-off: immutability or flexibility. There is always the opinion that smart contracts should not be made upgradeable [45]. One common compromise is the feature *time-delayed upgrades*, which is itself a trade-off.

7.2 Related Work

A prior work [35] studies the `CREATE2` usage, which includes some discussion on upgradeability, specifically about the *metamorphic contract* pattern. Another work [59] conducts a study on the access control of upgradeability in smart contracts. Rather than static analysis, it uses dynamic transaction traces to detect USCs. While this approach works without source code and could thus be promising, it does not work if a contract is not yet deployed or has few transactions. The authors further use this method to conduct a study primarily on 'who has admin rights to perform an upgrade'. In comparison, our research aims to answer three research questions with a more fine-grained view of real-world USCs.

In addition, the authors present an overview of upgradeability patterns, including a high-level overview of `delegatecall`-based patterns and different ways to avoid function selector clashes, as well as a number of other uncommon patterns which do not use a proxy. In contrast, our study focuses on the `delegatecall`-based patterns that have become the industry standard. By dissecting the complete set of elements that make up a smart

contract and identifying all elements which could possibly relate to proxy-based upgradeability, we build a taxonomy of proxy-based USCs and thus are able to detect a comprehensive set of upgradeability-related security issues via static analysis before a contract is deployed.

7.3 Limitations and Future Directions

With more adoption of multi-implementation patterns like Diamonds, UscHunt should do cross-contract analysis not only with one logic contract but with many. As UUPS proxies become prominent, it becomes even more important for UscHunt to reliably retrieve and analyze logic contracts with their proxies, even when the two use incompatible compiler versions.

Another limitation is UscHunt's inability to analyze closed-source contracts or those written in languages other than Solidity. For instance, Vyper is a Python-like language for smart contracts. Built atop Slither, UscHunt supports any language that it does, yet for now efforts to add Vyper support have stalled.

Beyond improving UscHunt, we believe there are other promising directions related to USC security, such as differential fuzzing between multiple logic versions. As Compound's token distribution bug demonstrates, upgrading a smart contract can introduce new bugs unrelated to any USC feature or pattern. Catching such bugs before an upgrade goes live can be critical.

8 Conclusion

In this work, we conduct the first large-scale study on proxy-based upgradeable smart contracts to uncover the status quo and report the related security issues. To do so, we develop a thorough taxonomy of proxy-based USCs that can uniquely characterize their behaviors based on both syntactic and semantic features. We further design and implement UscHunt, a novel static analysis framework for detecting and analyzing USCs. With it, we conduct the study with 800K+ smart contracts in eight mainstream blockchains. We report multiple important findings that include 11 unique USC design patterns and 6 different types of security and safety issues.

References

[1] Eip-1538: Transparent contract standard. `https://eips.ethereum.org/EIPS/eip-1538`, 2022.

[2] Eip-1822: Universal upgradeable proxy standard (uups). `https://eips.ethereum.org/EIPS/eip-1822`, 2022.

[3] Eip-1967: Proxy storage slots. `https://eips.ethereum.org/EIPS/eip-1967/`, 2022.

[4] Eip-2535: Diamonds, multi-facet proxy. `https://eips.ethereum.org/EIPS/eip-2535`, 2022.

[5] anchain.ai. anchain.ai: Last winner attack report. https://anchainai.medium.com/largest-smart-contract-attacks-in-block\\chain-history-exposed-part-1-93b975a374d0/, 2020.

[6] Arbitrum. https://arbitrum.io//, 2022.

[7] Avalanche. https://www.avax.network/, 2022.

[8] bnb. https://www.bnbchain.org/en, 2022.

[9] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 161–178. IEEE, 2022.

[10] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf, 2014.

[11] Celo. https://celo.org/, 2022.

[12] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 227–239. IEEE, 2021.

[13] Compound. Compound.finance. https://compound.finance/, 2022.

[14] ConsenSys. Mythril: security analysis tool for evm bytecode. https://github.com/ConsenSys/mythril/, 2022.

[15] CryptoverseCC. Ethmail.cc: Email services for ethereum community. https://ethmail.cc/, 2020.

[16] Daonomic. Daonomic/contracts-upgradeable. https://github.com/daonomic/contracts-upgradeable/, 2022.

[17] Michael del Castillo. The dao attacked: Code issue leads to $60 million ether theft. 2016.

[18] Yue Duan, Xin Zhao, Yu Pan, Shucheng Li, Minghao Li, Fengyuan Xu, and Mu Zhang. Towards automated safety vetting of smart contracts in decentralized applications. In *Proceedings of the 29nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2022.

[19] Ethereum. Ethereum smart contract document. https://ethereum.org/en/developers/docs/smart-contracts/, 2020.

[20] Ethereum. Solidity documentation: Layout of state variables in storage. https://docs.soliditylang.org/en/v0.8.15/internals/layout_in_storage.html, 2021.

[21] Ethereum. https://ethereum.org/en/, 2022.

[22] Etherscan. Arbitrum one explorer. https://arbiscan.io, 2022.

[23] Etherscan. Avalanche c-chain explorer. https://snowtrace.io, 2022.

[24] Etherscan. Binance smart chain explorer. https://bscscan.com, 2022.

[25] Etherscan. A block explorer and analytics platform for celo. https://celoscan.io, 2022.

[26] Etherscan. A block explorer and analytics platform for ethereum. https://etherscan.io, 2022.

[27] Etherscan. Fantom blockchain explorer. https://ftmscan.com, 2022.

[28] Etherscan. The optimism explorer. https://optimistic.etherscan.io, 2022.

[29] Etherscan. Polygon pos chain explorer. https://polygonscan.com, 2022.

[30] Fantom. https://fantom.foundation/, 2022.

[31] Yu Feng, Emina Torlak, and Rastislav Bodik. Precise attack synthesis for smart contracts. *arXiv preprint arXiv:1902.06067*, 2019.

[32] Christof Ferreira Torres, Antonio Ken Iannillo, and Arthur Gervais. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *European Symposium on Security and Privacy, Vienna 7-11 September 2021*, 2021.

[33] Christof Ferreira Torres and Hugo Jonker. Elysium: Context-aware bytecode-level patching to automatically heal vulnerable smart contracts. In *International Symposium on Research in Attacks, Intrusions and Defenses, Limassol, Cyprus 26-28 October 2022*, 2022.

[34] Joel Frank, Cornelius Aschermann, and Thorsten Holz. Ethbmc: A bounded model checker for smart contracts. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2757–2774, 2020.

[35] Michael Fröwis and Rainer Böhme. Not all code are create2 equal. In *6th Workshop on Trusted Smart Contracts (WTSC '22)*, 2022.

[36] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018.

[37] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. Online detection of effectively callback free objects with applications to smart contracts. In *Procceedings of The 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2018)*, 2018.

[38] Bo Jiang, Ye Liu, and Wing Kwong Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 259–269. IEEE, 2018.

[39] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: analyzing safety of smart contracts. In *Ndss*, pages 1–12, 2018.

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

[41] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68. IEEE, 2018.

[42] DeFi Llama. Defi llama - defi dashboard. `https://defillama.com`, 2021.

[43] looksrare. Looksrare nft marketplace. `https://looksrare.org/`, 2022.

[44] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269, 2016.

[45] Steve Marx. Upgradeability is a bug. `https://consensys.net/diligence/blog/2019/01/upgradeability-is-a-bug/`, January 2019.

[46] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://www.bitcoin.org/bitcoin.pdf`, May 2009.

[47] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 778–788, 2020.

[48] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th annual computer security applications conference*, pages 653–663, 2018.

[49] Trail of Bits. Contract upgrade anti-patterns. `https://blog.trailofbits.com/2018/09/05/contract-upgrade-anti-patterns/`, 2018.

[50] Trail of Bits. Manticore: symbolic execution tool for smart contract. `https://github.com/trailofbits/manticore/`, 2022.

[51] OpenSea. Opensea nft marketplace. `https://opensea.io/`, 2022.

[52] Optimism. `https://www.optimism.io/`, 2022.

[53] Martin Ortner and Shayan Eskandari. Smart contract sanctuary.

[54] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*, pages 1661–1677. IEEE, 2020.

[55] Polygon. `https://polygon.technology/`, 2022.

[56] Valuates Report. Smart contract market size. `https://reports.valuates.com/market-reports/QYRE-Auto-31L1599/global-smart-contracts/`, 2021.

[57] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. In *Proceedings of the 2019 Network and Distributed System Security Symposium*, 2019.

[58] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. Evmpatch: timely and automated patching of ethereum smart contracts. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[59] Mehdi Salehi, Jeremy Clark, and Mohammed Mannan. Not so immutable: Upgradeability of smart contracts on ethereum. In *6th Workshop on Trusted Smart Contracts (WTSC '22)*, 2022.

[60] Evgeniy Shishkin. Debugging smart contract's business logic using symbolic model checking. *Programming and Computer Software*, 45(8):590–599, 2019.

[61] Slither. Slither, the solidity source analyzer. `https://github.com/crytic/slither/`, 2022.

[62] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. Verismart: A highly precise safety verifier for ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1678–1694. IEEE, 2020.

[63] Solidity. Solidity documentation. `https://docs.soliditylang.org/en/v0.8.16/`, 2022.

[64] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. Smartpulse: automated checking of temporal properties in smart contracts. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 555–571. IEEE, 2021.

[65] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 9–16, 2018.

[66] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 664–676, 2018.

[67] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, 2018.

[68] Uniswap. Uniswap protocol. `https://uniswap.org/`, 2022.

[69] Yuepeng Wang, Shuvendu K Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, and Immad Naseer. Formal specification and verification of smart contracts for azure blockchain. *arXiv preprint arXiv:1812.08829*, 2018.

[70] Valentin Wüstholz and Maria Christakis. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1398–1409, 2020.

[71] Valentin Wüstholz and Maria Christakis. Targeted greybox fuzzing with static lookahead analysis. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 789–800. IEEE, 2020.

[72] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1029–1040, 2020.

[73] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. xfuzz: Machine learning guided cross-contract fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[74] ZenLedger. Compound's defi bug. `https://www.zenledger.io/blog/compounds-defi-bug-what-happened-how-to\\-handle-erroneous-transfers/`, 2021.

[75] Zeppelin. Openzeppelin: Upgrading smart contracts. `https://docs.openzeppelin.com/learn/upgrading-smart-contracts/`, 2022.

[76] Zeppelin. Openzeppelin: Writing upgradeable contracts. `https://docs.zeppelinos.org/docs/2.1.0/writing_contracts`, 2022.

[77] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. Smartshield: Automatic smart contract protection made easy. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 23–34, 2020.

A Appendix

Table 4: Low-Level Syntactic Features Related to Upgradeability

Related Contract Elements	Feature Set	Syntactic Features	Possible Values
F	*fallback* function	Contains `delegatecall`	Y / N
		Reverts for specific address	Y / N
		Loads target address from hard-coded storage slot	Y / N
		`delegatecall` dominated by conditional check	Y / N If yes, is condition modifiable?
		Checks function selector in calldata before loading target address	Y / N
V, I, T	`delegatecall` target *addr$_{logic}$*	Definition location	Proxy contract Proxy and logic contracts External contract
		Type	address type contract type bytes32 type (constant storage slot) address array mapping(__ =>address)
		Scope	State variable Local variable Structure variable Literal / constant variable
		Inheritance	Inherited by proxy Inherited by proxy and logic None
F, I	*setter*	Definition location	Proxy contract Logic contract Inherited contract External contract
		Timestamp-related revert	Y / N
		Writes to storage slot using `sstore`	Y / N
F, I	*getter*	Definition location	Proxy contract Logic contract Inherited contract External contract
		Contains external call	Y / N
		Reads from storage slot using `sload`	Y / N
F	External functions	External functions in proxy other than fallback/receive	Y / N
		External functions in proxy unrelated to upgradeability	Y / N
		External functions require specific caller	Y / N
		External functions contain `delegatecall`	Y / N
V	Other state variables	Mappings for each variable type	Y / N
		Stores contract address other than *addr$_{logic}$*	Y / N
		Stores timestamp variable to check in *setter*	Y / N
		Stores conditional variable to check in *fallback* function	Y / N
		Stores admin address variable for access control	Y / N

(a) Arbitrum   (b) Avalanche   (c) BSC

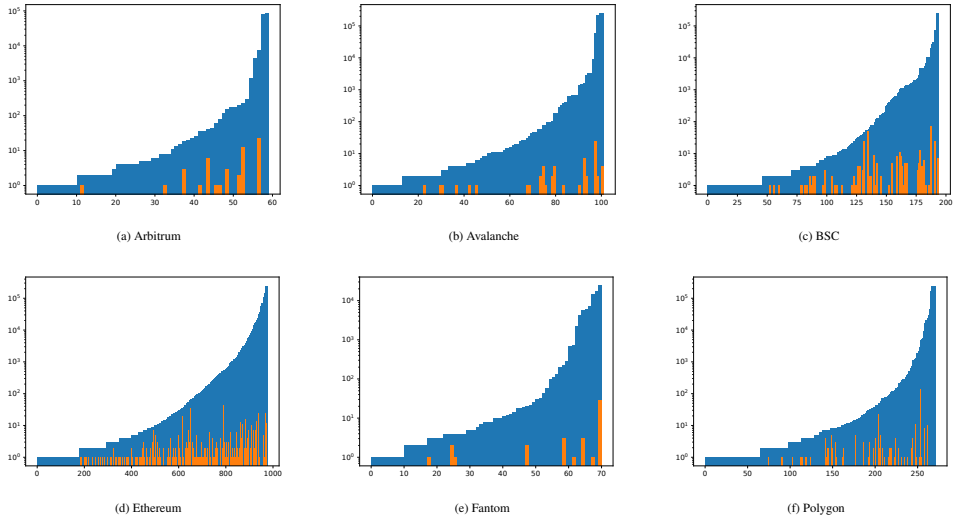(d) Ethereum   (e) Fantom   (f) Polygon

Figure 8: Upgrade Frequency: setter calls (orange) compared to total transaction count (blue)

Table 5: High-level Semantic Features

Category	Semantic Features	Possible Values	Related Syntactic Features
Data Definition and Storage	Constant storage offset	Y / N If yes, which slot? What is the hashed string?	*addr_logic* type: bytes32 constant storage slot *setter* writes to storage slot using sstore: yes *getter* reads from storage slot using sload: yes
Data Definition and Storage	Storage layout coupling	Y / N	*addr_logic* definition location: proxy and logic contracts (separate definitions) or inherited contract (shared by proxy and logic) *addr_logic* scope: state variable
Upgradeability Implementation	Simultaneous upgrades	Y / N	*addr_logic* definition location: external contract *setter* definition location: same external contract *getter* definition location: same external contract *getter* (or fallback function) contains external call: yes
Upgradeability Implementation	Scattered implementations	Y / N	*addr_logic* type: mapping(bytes4 =>___) *addr_logic* scope: state variable or structure variable Fallback function checks function selector before loading target: yes
Upgradeability Implementation	Partially upgradeable	Y / N	External functions in proxy other than fallback/receive: yes External functions contain delegatecall: optional
Upgradeability Implementation	Transparent admin check	Y / N	External functions require specific caller address: yes Fallback function reverts for specific caller address: yes, same address
Modifiability	Time-delayed upgrades	Y / N If yes, how long?	Stores timestamp variable to check in *setter*: yes *setter* has timestamp-related revert: yes
Modifiability	Toggleable delegatecall	Y / N	delegatecall in fallback function dominated by conditional check: yes, and condition is modifiable
Modifiability	Removable upgradeability	Y / N If yes, how?	*setter* definition location: logic contract or external contract (external contract address must not be constant)

Table 6: USC Design Pattern Definition

	Pattern Name	Inherited Storage	Eternal Storage	Unstructured Storage			Mastercopy / Singleton	Transparent Proxy	Multiple Implementations		Registry Proxies	
	Sub-Pattern	-	-	Non-standard	EIP-1967: Standard Storage Slots	EIP-1822: UUPS	-	-	EIP-1538: VTable	EIP-2535: Diamond	Beacon	Registry
Syntactic Features	Target location	Inherited contract	-	Proxy contract	Proxy contract	Proxy and logic	Proxy and logic	Proxy contract	Proxy contract	Proxy contract	External contract	External contract
	Target type	-	-	bytes32	bytes32	bytes32	address	-	mapping (bytes4 => address)	mapping (bytes4 => Facet struct)	address	mapping (___ => address)
	Target scope	State variable	State variable	Constant	Constant	Literal in fallback	State variable	-	State variable	Structure variable	State variable	State variable
	Target inheritance	Inherited by proxy and logic	-	-	-	Logic must inherit Proxiable	-	-	-	-	-	-
	Setter location	-	-	Proxy contract	Proxy contract	Logic contract	Logic contract	Proxy contract	Proxy contract	Logic contract	External contract	External contract
	Getter location	-	-	Proxy contract	Proxy contract	Proxy contract (fallback)	Proxy contract (fallback)	Proxy contract	Proxy contract (fallback)	Proxy contract (fallback)	External contract	External contract
	Mappings of each type	-	Yes	-	-	-	-	-	-	-	-	-
Semantic Features	Constant storage offset	No	No	Yes, slot varies	Yes, hashed string is eip1967.proxy.implementation	Yes, hashed string is PROXIABLE	No	-	No	No, but struct may be stored in storage slot	No, but beacon address may be stored in storage slot	No, but registry address may be stored in storage slot
	Storage layout coupling	Yes	Yes	No	No	No	Yes	-	-	-	-	-
	Simultaneous upgrades	-	-	-	-	-	-	-	-	-	Yes	Yes
	Removable upgradeability	-	-	No	No	Upgrade to logic w/o setter	Upgrade to logic w/o setter	-	-	Remove Diamond CutFacet	Update to beacon w/o setter	Update to registry w/o setter
	Transparent admin check	-	-	-	-	-	-	Yes	-	-	-	-
	Scattered implementations	-	-	-	-	-	-	-	Yes	Yes	-	-

Table 7: Detected USC-related Security Issues

	Implementation Issues				Policy Issues	
Chain	Storage Layout Clashes (Between Proxy and Logic)	Storage Layout Clashes (Between Logic Versions)	Function Selector Collisions	Insufficient Compatibility Checks In Logic Setter	Vulnerabilities Cannot Be Patched Immediately	Upgradeability Can Be Removed Accidentally
Ethereum	36	3	0	1,017	24	150
Arbitrum	1	0	0	87	0	12
Avalanche	1	1	0	137	3	12
BSC	7	0	0	337	0	23
Celo	0	0	0	25	0	0
Fantom	3	0	0	166	0	2
Optimism	0	0	0	33	0	5
Polygon	8	0	0	441	3	9
Total	56	4	0	2,243	30	213