Wielding Blockchain Transactions for Capture-Replay Testing of Upgradeable Smart Contracts

MORENA BARBONI, Computer Science, University of Camerino, Camerino, Italy
GUGLIELMO DE ANGELIS, IASI-CNR, Rome, Italy
ANDREA MORICHETTA, Computer Science, University of Camerino, Camerino, Italy
ANDREA POLINI, Computer Science, University of Camerino, Camerino, Italy

Blockchain technology is increasingly adopted in scenarios requiring trust and data integrity. On the Ethereum blockchain, the proxy pattern has become increasingly popular because it allows smart contract code to evolve while preserving stored data. However, a key challenge remains ensuring that such upgrades do not introduce breaking changes or cause disruptions to other contracts and off-chain systems. In this article, we introduce CATANA, a framework that leverages historical transactions for Capture-Replay testing of proxy-based Upgradeable Smart Contracts (USCs). CATANA assesses the potential impact of an upgrade by comparing the outcomes of replayed transactions with those from the previous version deployed on the main network. Additionally, it extracts and decodes contract state variables, providing deeper insights into how code changes affect the contract state, and helping developers mitigate issues before deployment. Experiments demonstrate that analyzing storage data accounts for the majority (about 86.5%) of detected disruptive upgrades. We also evaluate different policies for building replay test suites from historical transactions. Results identify a strategy that maximizes effectiveness while requiring a small number of replay test executions. Even a test suite containing just one transaction per each invoked method achieved good effectiveness (about 60%) in detecting disruptive upgrades.

CCS Concepts: • **Software and its engineering → Software testing and debugging;**

Additional Key Words and Phrases: Upgradeable smart contracts, capture-replay testing, blockchain, ethereum

ACM Reference Format:
Morena Barboni, Guglielmo De Angelis, Andrea Morichetta, and Andrea Polini. 2025. Wielding Blockchain Transactions for Capture-Replay Testing of Upgradeable Smart Contracts. *ACM Trans. Internet Technol.* 25, 3, Article 19 (August 2025), 30 pages. https://doi.org/10.1145/3737699

This research was carried out within the framework of the Fermo Tech Extended project, funded by the Department for Territorial Cohesion, Presidency of the Council of Ministers, Italian Government, project code E77G23000130001. This research was also partially supported by the project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by NextGenEu, and partially supported by the Italian MUR PRIN 2022 Project: Domain (Grant Agreement #2022TSYYKJ) funded by NextGenEu. Guglielmo De Angelis is with the Italian Research Group: INdAM-GNCS..
Authors' Contact Information: Morena Barboni, Computer Science, University of Camerino, Camerino, Macerata, Italy; e-mail: morena.barboni@unicam.it; Guglielmo De Angelis, IASI-CNR, Rome, Lazio, Italy; e-mail: guglielmo.deangelis@iasi.cnr.it; Andrea Morichetta, Computer Science, University of Camerino, Camerino, Macerata, Italy; e-mail: andrea.morichetta@unicam.it; Andrea Polini, Computer Science, University of Camerino, Camerino, Macerata, Italy; e-mail: andrea.polini@unicam.it.

ACM 1533-5399/2025/08-ART19
https://doi.org/10.1145/3737699

1 Introduction

Blockchain technology is widely known for its ability to store data immutably, ensuring that once data is recorded, it cannot be altered in any practical manner. Among the existing blockchain platforms, Ethereum stands out due to its support for smart contracts, tamper-proof programs that can be deployed and executed on the blockchain [40]. This capability allows users to interact with **decentralized applications** (**DApps**) with confidence, knowing that the underlying logic cannot be arbitrarily modified. Although the immutability of smart contracts fosters integrity and trust, it also presents a significant challenge: it prevents direct bug fixes and feature updates. To address this limitation, developers have introduced mechanisms for implementing **Upgradeable Smart Contracts** (**USCs**), with the **proxy pattern** emerging as a popular solution [10, 38]. This pattern decouples a contract's storage from its logic, allowing developers to release upgrades without redeploying or migrating stored data. Thanks to its convenience and simple design, the proxy pattern has since been formalized through community standards and is now supported by major frameworks such as OpenZeppelin [32].

While the proxy pattern is now considered technically mature, developers still approach upgrades with caution. The fact that only a small percentage of USCs have been updated more than once indicates concerns about the potential impact of the introduced changes [34]. One key contributing factor is the lack of proper support for regression testing. Developers primarily upgrade contracts to improve usability, typically by adding or modifying functionalities [28]. However, beyond the risk of introducing new bugs or security vulnerabilities [27], changes to the business logic of a smart contract can cause *compatibility issues*. A large-scale study over 60M Ethereum smart contracts revealed frequent occurrences of **Application Binary Interface** (**ABI**)-breaking changes in proxy-based systems [27], where updates to the logic contract break compatibility with the external interface. Another problem is given by the potential for storage collisions [23, 36], where new variables in the logic contract overlap with existing storage slots. Both issues are highly relevant, but more subtle changes that neither break the ABI nor corrupt storage can still affect smart contract behavior while being harder to detect. Such changes may introduce inconsistencies in contract outputs and states, possibly causing disruptions in dependent systems (e.g., in DApp front-ends, off-chain systems, or other smart contracts), and leading to unintended consequences for users. Current tools offer only partial safeguards, as they mostly focus on preventing storage collisions. For example, OpenZeppelin's Upgrades plugin [33] includes useful pre-deployment checks to ensure storage integrity, which alone does not guarantee behavioral compatibility. To truly improve the reliability of USCs, more advanced testing approaches are needed. Fine-grained behavioral analysis, particularly one that accounts for real-world usage patterns, is essential to detect subtle but disruptive changes before upgrades reach production.

Capture-Replay testing is a regression testing technique that collects interactions between users and the **system under test** (**SUT**), leveraging them to create replayable test scripts [7, 8, 21, 24, 26]. While capturing execution traces in traditional systems can be intrusive and introduce performance overhead, public blockchains immutably and publicly log all user transactions by design. This offers a unique opportunity for non-intrusive, large-scale replay testing of blockchain applications. However, despite this potential, no existing framework is designed to detect behavioral incompatibilities introduced by smart contract upgrades. Prior work has explored historical transactions in other contexts, but with fundamentally different objectives. For example, SmartGift [45] demonstrates how real-world transactions can guide the generation of effective test cases, but it focuses on newly created contracts similar to the ones deployed, not regression testing of upgraded ones. Other transaction-centric frameworks emphasize monitoring, attack detection, or runtime protection [12, 41, 43], without addressing upgrade compatibility.

This work presents the first regression testing approach and framework for Ethereum Proxy-based USCs using historical transaction replay. The approach consists of four main steps: *(i)* retrieving transactions executed on a deployed USC to construct a replay test suite; *(ii)* forking the Ethereum Mainnet and executing these transactions on the original USC to derive test oracles (i.e., the expected outcomes); *(iii)* simulating an upgrade and replaying the transactions on the new USC version; and *(iv)* verifying the consistency of outcomes. We implemented this procedure in a reference framework called **CATANA** (***Contract Assessment through TrANsaction replAy***), which we made open-source on GitHub. Catana is designed to support smart contract testers in *identifying disruptive changes* prior to deployment, *validating intentional changes*, and *debugging* eventual issues by analyzing pre- and post-upgrade logs. Catana is agnostic to the proxy pattern used, making it applicable to established upgradeability standards (e.g., Transparent and UUPS Proxies [32]). This article builds on the results of our previous research [4] and expands them in several ways. First, we extend Catana's replay testing workflow to include the extraction of all state variables stored within a USC. This snapshot allows Catana to analyze not just the output of a replayed transaction but also its impact on the internal state of the proxy. This improves observability during testing and offers deeper insights into how contract upgrades affect both behavior and state. Furthermore, considering the high volume of transactions typically associated with real-world USCs, we evaluate the effectiveness of four selection policies for building replay test suites, exploring the tradeoffs between test suite size and fault detection capabilities. In particular, this study is driven by the following **research questions** (**RQs**):

— **RQ1**: *How does storage data analysis impact the effectiveness of Capture-Replay testing for USCs?*
— **RQ2**: *Which transaction selection policy most effectively identifies disruptive behaviors in USCs?*

The experimental results show that Capture-Replay testing based on historical transactions can be a valid complement to traditional testing practices. Storage data analysis proved essential for flagging code changes that might affect the behavior of a USC, potentially leading to regression issues and compatibility concerns. Indeed, out of all the disruptive upgrades that Catana detected, about 86.5% were exclusively identified through changes in state variables. Furthermore, our evaluation of test selection policies shows that focusing on method-level diversity consistently outperforms other approaches in terms of fault-detection and the number of replayed transactions. Notably, even a test suite containing just one transaction per each invoked method achieved good effectiveness, detecting about 60% of disruptive upgrades.

Overall the main contributions of this article can be summarized as follows:

— A novel capture-replay testing approach for proxy-based USCs based on historical Ethereum transactions;
— The refinement of Catana, an open-source and proxy-agnostic framework for capture-replay testing, which now supports the detection of potentially disruptive changes in upgrades, validation of intentional changes, and debugging;
— An empirical assessment that estimates the impact of observability during replay testing through both output comparison and storage state analysis;
— The evaluation of four transaction selection policies, highlighting the tradeoffs between the size of the generated replay test suites and their fault-detection capabilities;
— The formulation of a set of guidelines for practitioners resuming the main insights we observed in the study.

The rest of this article is organized as follows: Section 2 provides key concepts about Ethereum, USCs, and replay testing. Section 3 describes the design of Catana and its replay testing worklow

in detail. Section 4 expands on the RQs, while Sections 5 and 6 describe the experiment setup and results. Section 7 discusses threats to validity of the study. Finally, Section 8 reviews related work and Section 9 identifies future research directions.

2 Background

In the next section, we provide background knowledge about the Ethereum blockchain, covering key concepts such as the account model and transactions. In Section 2.2, we present smart contracts, detailing their storage model and existing upgradeability mechanisms, with a particular focus on the proxy pattern. In Section 2.3, we provide an overview of the Capture-Replay testing technique and its practical applications. Lastly, in Section 2.4, we briefly introduce mutation testing, as it is relevant to the validation of our approach.

2.1 Ethereum

Ethereum is a decentralized, public blockchain platform that supports the execution of smart contracts [11], self-enforcing agreements that operate without the need for intermediaries. This versatility enables developers to build a wide array of distributed applications, from finance to gaming, within a secure and trustless environment.

2.1.1 Accounts. The Ethereum blockchain features two primary types of accounts:

(1) **Externally Owned Accounts (EOAs)** are user-controlled gateways to the Ethereum network, managed through private keys. They are used to exchange Ether (ETH) and interact with smart contracts via *transactions*.
(2) **Contract Accounts (CAs)** represent *Smart Contracts* deployed to the Ethereum network, and function as autonomous executors of decentralized logic. Each smart contract has a unique public address, an ETH balance, bytecode, and storage. The *bytecode* is stored on the blockchain and executed by the **Ethereum Virtual Machine** (**EVM**) upon invocation, enabling CAs to perform predefined functions. The *storage*, organized as a Key-Value map, holds the contract's global values and is accessed or modified by the EVM during execution.

2.1.2 Transactions. Ethereum transactions are the fundamental operations that drive interactions on the blockchain, and can be categorized into three main types:

(1) A **Regular Transaction** involves an EOA sending ETH to another EOA;
(2) A **Contract Creation** allows an EOA to deploy smart contract code and create a new CA with its unique address.
(3) A **Contract Execution** allows an EOA to invoke and execute a method of a CA. This process involves running the bytecode of the smart contract, possibly reading from or writing to the contract's *storage*.

Given our focus on capture replay testing, we are only interested in **contract executions**, as they permit us to test and validate smart contract behavior under various scenarios.

2.1.3 Ethereum World State. The Ethereum world state tracks all accounts and their states, including balances, storage, and code. The world state is updated after each transaction, reflecting the changes in account states resulting from the transaction's execution. In particular, a transaction can modify the state of EOAs by adjusting their Ether (ETH) balances or altering the **storage** and code associated with CAs. The EVM executes the transaction's instructions and ensures that all changes are consistent with the rules defined by the smart contract bytecode and the overall Ethereum protocol. The resulting changes are recorded in the world state and committed to the blockchain as part of a new block, providing a *historical record* of *all state changes* over time. In relation to the world state, smart contract development and testing tools such as Hardhat introduce a

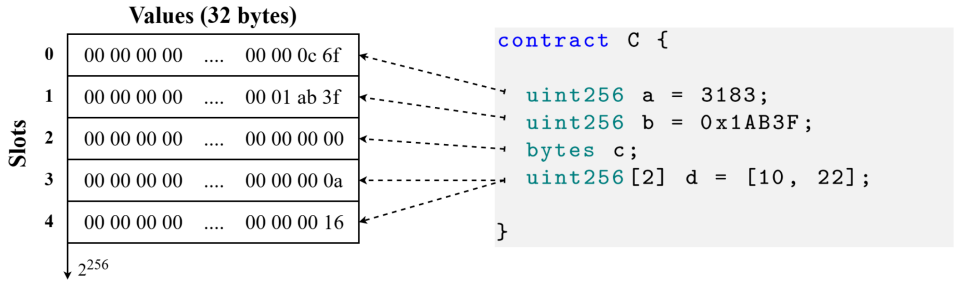


Fig. 1. Example of smart contract storage layout.

powerful concept known as **forking**.[1] Forking involves creating a local instance of the Ethereum blockchain that mirrors the *world state* of the Mainnet (or any other network), including all account balances, contract storage, and deployed code. To fork a network, the testing tool connects to an Ethereum node and retrieves the state of the blockchain at a specified block. This state is then replicated in a local network (e.g., the Hardhat Network). This allows developers to deploy and test smart contracts in a production-like environment without interacting with the Mainnet or incurring transaction costs.

2.2 Smart Contracts

Smart contracts are programs that automatically execute the terms of an agreement when predefined conditions are met, enabling trustless interactions without the need for intermediaries. They are written in a high-level, contract-oriented programming language like *Solidity*, which is then compiled into bytecode and deployed on the blockchain. Code immutability guarantees that the business logic implemented by a contract cannot be altered after its deployment. To allow for bug fixes and feature upgrades, several upgradeability patterns have been introduced, enabling changes to the contract's logic while preserving its original address and storage. In the following, we will present the storage model and the main characteristics of USCs.

2.2.1 The Smart Contract Storage Model. Every Ethereum smart contract features a permanent **storage** that retains data between transactions. This storage functions as an independent, read-write memory area holding the values of the contract's **state variables**. The storage acts like a *key-value database*, where each key corresponds to a storage **slot** number p, and the **value** represents the content of that slot. The way Ethereum organizes this data is called the **storage layout** (**SL**), which follows a structured approach but can become complex, especially for advanced data types (see the Solidity documentation[2]). As shown in Figure 1, each state variable declared in a smart contract is typically assigned a slot p in storage sequentially. But for dynamic or complex data types (e.g., dynamic arrays, mappings, and structs) additional rules must be applied to determine where their data is stored. Dynamic arrays, for instance, store the number of elements in a slot, while the actual data is placed elsewhere, found using a cryptographic hash function. Mappings (used for key-value associations like user balances) don't have fixed storage locations for their elements. Instead, each entry's position is calculated dynamically based on its key. Utilities like `hardhat-storage-layout`[3] support developers by displaying information about a contract's state variables and how they are laid out in storage. This can reduce the risk of human

[1]Hardhat forking: https://hardhat.org/hardhat-network/docs/guides/forking-other-networks
[2]Layout of State Variables in Solidity: https://docs.soliditylang.org/en/latest/internals/layout_in_storage.html
[3]Hardhat Storage Layout: https://www.npmjs.com/package/hardhat-storage-layout

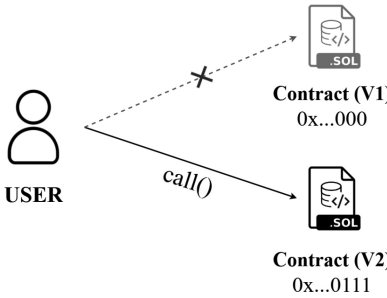


Fig. 2. Non-upgradeable smart contract.

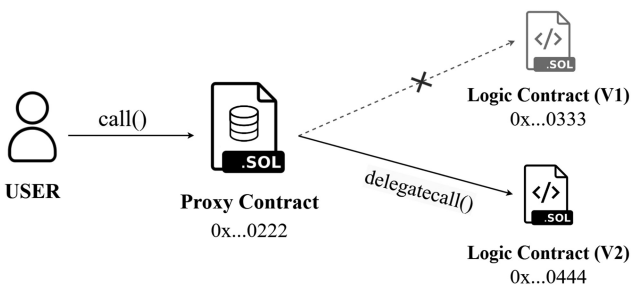


Fig. 3. Upgradeable smart contract using the proxy pattern.

error when working with smart contracts and updating specific storage slots. However, these utilities only display the initial slot p where each state variable is located, offering a limited view of the storage data. Advanced testing techniques that involve the analysis of state variables require more comprehensive methods to thoroughly explore the storage content of a smart contract.

2.2.2 Upgradeable Smart Contracts (USCs) and the Proxy Pattern. The main characteristic of a smart contract is its immutability, meaning its source code cannot be directly modified after deployment. In the case of a traditional (i.e., non-upgradeable) smart contract (Figure 2), the code can only be updated through a *contract migration* [27]. This is a process where a new contract (i.e., the new version) is deployed at a different address, starting with an empty storage. The state data from the original contract (i.e., the old version) is then migrated to the new one. Usually, an official announcement follows to make the community aware of the migration, allowing users and dependent systems to transition to the new address. This approach is not only inconvenient but also introduces complexities and costs due to the data migration process. Upgrading a smart contract means switching the code executed at a given address while preserving its state, and without disrupting user interactions. USCs can be built using different patterns (e.g., the data separation pattern, the strategy pattern, and metamorphic contracts [27]), however the *proxy pattern* has gained significant popularity due to its effectiveness and simple design. A study on smart contract upgradeability [38] shows that proxy-based projects are widespread on the Ethereum main chain, with 1.4 million proxy contracts used in more than 8k upgradeable systems. Etherscan [14], the largest analysis and statistics platform dedicated to Ethereum, has also added automated verification for proxy contracts, making it more intuitive for users to know which implementation contract they are interacting with.

The idea behind the proxy pattern is to *decouple the storage* of a smart contract *from its logic*. This way, clients always interact with the same proxy contract, while the underlying logic can be replaced without losing any storage data. As illustrated in Figure 3, a basic proxy architecture includes two components: a **proxy** contract and a **logic** contract (sometimes also referred to as *implementation* contract). All transactions issued by users target the proxy, which maintains the relevant storage data and balance of the DApp, as well as a mutable reference (i.e., an address) to a specific logic contract. When the proxy is invoked, it automatically reroutes the transaction to the referenced logic contract, which implements the DApp's business logic. This logic can therefore be replaced without re-deploying the proxy and incurring in costly state migrations. To achieve storage decoupling, the proxy pattern relies on two key mechanisms. Each transaction to the proxy is redirected to the logic contract using a `delegatecall`. This is a low-level Ethereum function that executes the logic of the called contract in the context of its caller. This means any operation performed by the logic contract will only affect the state of the proxy. Beyond the delegatecall, proxy

contracts also require a function for controlling upgrades, allowing the address of the implementation used by the proxy to be changed with the address of a different logic contract. Generally, the upgrade function is built directly into the proxy with appropriate access control mechanisms to prevent unauthorized users from making changes.

2.3 Capture-Replay Testing

Capture-Replay testing (also known as Record-Replay) is a technique used to verify the behavior and functionality of software systems by replaying previously captured inputs. The approach comprises two main phases: During the **Capture** phase, the testing tool records all interactions between users and the SUT. This includes carving detailed information about the inputs provided by users, the corresponding outputs generated by the system, and the context of these interactions [9]. The data collected typically encompasses input parameters for function calls, HTTP requests and responses in web applications [1, 29], or user gestures and touch events in mobile applications [7, 17, 35]. The granularity of this captured data can vary based on the requirements, ranging from high-level user actions to low-level system calls. In the **Replay** phase, the captured data is transformed into executable test scripts that simulate the previously recorded real-world interactions. Replay tests aim to detect unexpected system behaviors, such as regressions, errors, or performance issues that may have been introduced after system changes or updates. By comparing the outputs during the replay with the expected results (as recorded during the capture phase), testers can identify deviations and potential defects in the software. Indeed, an important application of replay testing is in *regression testing* [2, 21]. Regression testing involves retesting a software system to ensure that code changes have not introduced new defects or negatively impacted existing functionality. By recording online traffic during normal operations, replay testing can build a robust suite of regression tests without requiring a deep knowledge of the system. Furthermore, since these tests are based on real usage patterns, they tend to be more realistic and effective in uncovering issues that might not be caught by traditional testing methods. However, one of the challenges of replay testing is managing the potentially large volume of recorded interactions. In many cases, the amount of data captured can be overwhelming, with a significant portion of the interactions being redundant or less critical for the testing objectives. For instance, repeated or similar user actions may not add much value to the replay process and can lead to inefficiencies in test execution. Therefore, it is essential to establish criteria for selecting the most relevant interactions to replay.

2.4 Mutation Testing

Mutation testing is a well-established technique for evaluating the quality of software tests. It works by introducing small, artificial bugs (called *mutants*) into the source code and then checking whether the existing tests can catch them. The ratio of mutants that are successfully detected, known as the ***Mutation Score*** (**MS**), indicates the test suite's fault-detection capabilities. Mutation testing operates under the fundamental hypothesis that if a test suite can identify simple faults mimicked by mutants, it is also likely to catch more complex, compound faults. Studies have demonstrated a strong correlation between the ability to detect mutants and the capacity to uncover real-world bugs [25], making this technique particularly valuable in high-stakes domains like blockchain. Solidity, being the main language for developing Ethereum smart contracts, has attracted significant research into mutation testing. Tools like *SuMo* [3, 5] inject a variety of mutations into the contract's code, covering Solidity-specific programming aspects such as the *address* data type, *transaction reverting* statements, *global blockchain variables*, and special functions for *transferring funds*. The effects of a mutant on a smart contract can be varied and manifest in different ways. A change in logic might cause a function to return different results, a

previously successful transaction to fail, or an operation to alter storage in ways that diverge from the original execution. For instance, replacing a literal numerical value in a contract's logic may affect balance calculations and other critical behaviors, while altering return values can directly impact the observable output of a function. Deleting a function modifier may remove an access control restriction, allowing transactions that would otherwise be rejected to modify storage. Past empirical studies demonstrate the value of Solidity mutants in evaluating test suites [5], confirming the Mutation Score as a dependable quality indicator in the smart contract domain.

3 The Catana Framework

Upgrading smart contracts deployed on the Ethereum Mainnet is a delicate activity. Developers must be aware of how code changes impact contract behavior, whether the new code performs as expected and whether it might break existing functionalities. To illustrate how Catana assists in upgrade scenarios, let's consider a proxy-based USC including a Proxy contract P and a Logic contract L_n. The proxy features several transactions as it is being actively used by the community. Over time, the logic contract L_n must be upgraded to L_{n+1} to improve some functionalities or to fix bugs. Catana is designed to ensure the consistency of this upgrade process by verifying that selected transactions executed on $\text{USC}(P, L_n)$ give the same output on $\text{USC}(P, L_{n+1})$. Moreover, Catana logs any changes in the smart contract storage that occur as a result of the upgrade. These logs provide developers with a detailed record of any storage modifications, allowing them to assess the impact of the code changes. Overall, Catana can support developers in the following ways:

(1) **Identify Disruptive Changes**: Catana flags any discrepancy in transaction outputs or storage values. This makes developers aware of potentially disruptive changes and allows them to review code before deployment.
(2) **Validate Intentional Changes**: When the upgrade is meant to alter certain behaviors (e.g., after introducing a bug fix), Catana's logs confirm whether the modifications align with the developer's expectations.
(3) **Support Contract Analysis and Debugging**: The pre and post-upgrade state logs allow developers to more easily understand the impact of the modifications and determine the cause of eventual issues.

The tool is open source and can be found on GitHub.[4] In the following section, we describe the replay testing process of Catana, while Section 3.2 provides details its storage analysis workflow.

3.1 Capture-Replay Testing Workflow

Catana automates Capture-Replay testing for any proxy-based USC. It captures transactions from a deployed $\text{USC}(P, L_n)$ and harnesses them to validate an upgraded $\text{USC}(P, L_{n+1})$, without having to manually create or configure test scripts. To this end, Catana requires access to:

— *Transactions to be replayed*: a set of transactions T that were executed on $\text{USC}(P, L_n)$;
— *Deployed USC addresses*: The addresses of $\text{USC}(P, L_n)$, as deployed on the Mainnet;
— *Upgraded USC sources*: The source code and ABI of the $\text{USC}(P, L_{n+1})$ under test.

Although the upgraded sources are assumed to be available locally, Catana automatically retrieves the addresses and historical transactions of $\text{USC}(P, L_n)$ by interacting with external services such as Etherscan.[5] Once these inputs are gathered, Catana sets up the test environment and begins the replay process. Under the hood, Catana leverages the Hardhat[6] framework and

[4]Catana repository: https://github.com/MorenaBarboni/Catana
[5]Etherscan API: https://etherscan.io/apis
[6]Hardhat: https://hardhat.org/

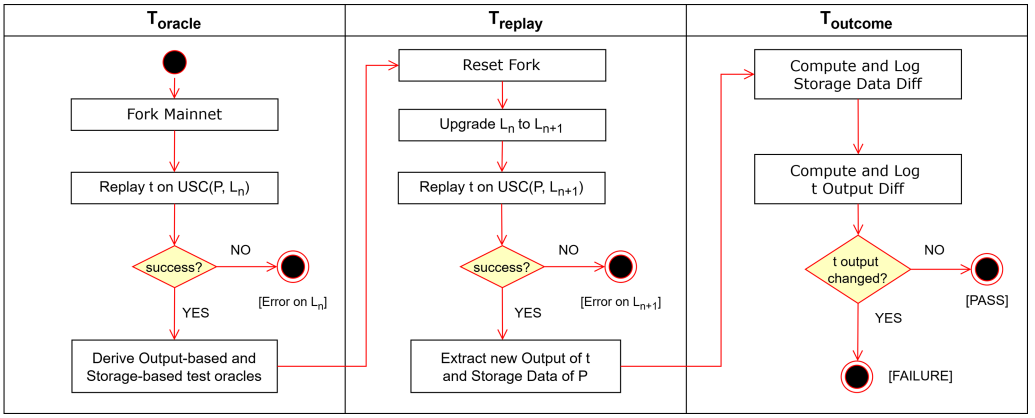


Fig. 4. Capture-Replay testing workflow as implemented in CATANA.

encapsulates its replay testing logic in a single custom test file. This file defines three parametric test methods, each targeting one transaction $t \in T$. The first method, T_{oracle}, runs the transaction t on USC(P, L_n) to derive the test oracles and stores them for future comparison. T_{replay} simulates the logic upgrade to USC(P, L_{n+1}) and replays t under the upgraded contract's logic, capturing the new test outcomes. Lastly, T_{outcome} checks the new outcomes against the oracles, reporting any discrepancies that might indicate regression bugs or alterations in behavior. The testing workflow is illustrated in Figure 4, and in the following we describe each step.

Step (1) T_{oracle}. This test method automatically derives oracles for regression testing based on the behavior of the original USC(P, L_n). In particular, it captures:

— **Output-based test oracle**: the return value produced during a transaction t (if any);
— **Storage-based test oracle**: a snapshot of the contract's state variables after the transaction t completes.

To achieve this, the test begins by forking the Ethereum Mainnet at the precise block right before the transaction of interest t occurred. This ensures that the contract's state matches the one at the time of the original execution. By forking Mainnet, Hardhat retrieves the historical on-chain data and exposes it as though it were available locally, eliminating the need to mock external dependencies that might be invoked by t. Once the environment is forked, the test obtains the proxy contract (P) and logic contract (L_n) from the forked network using their respective ABI definitions and deployed addresses. It then impersonates t's original sender address and replays the transaction on USC(P, L_n), applying the historical inputs and value (i.e., the amount of transferred funds). This replay involves two types of calls: (i) a *static call*, which simulates execution without modifying the blockchain state and allows CATANA to capture the *output-based* oracle (including any return values or error messages), and (ii) a standard *transaction call*, which actually modifies the state. As shown in Figure 4, error scenarios are handled to ensure meaningful replays. If a call fails on the original contract (e.g., due to reverts, network time-outs, or unexpected errors) it is excluded from further replay testing. If the transaction call completes, CATANA records the *storage-based oracle* by extracting state variables from the forked environment. Further details on how these variables are accessed can be found in Section 3.2.

Step (2) T_{replay}. The second test method upgrades the forked environment so that USC(P, L_n) now uses L_{n+1} as its implementation. Typically, a smart contract upgrade involves deploying the new logic contract on the network and updating P to reference it. However, this procedure can

be complicated by voting protocols and by differences in the specific upgrade pattern used. For example, the Transparent Proxy Pattern requires an external admin contract to execute an upgrade function, whereas other standards (e.g., the UUPS) do not have such a requirement. Rather than relying on upgrade-specific execution paths, CATANA performs an "in-place" upgrade by directly replacing the bytecode at L_n's address with that of L_{n+1}. Once this upgrade is performed, T_{replay} replays the same transaction t on the upgraded contract, extracting the new output and the new state snapshot. These will be compared against the baselines obtained by T_{oracle}. Note that if a transaction that succeeded on the original contract fails on the upgraded contract, the nature of the failure is taken into account. Unexpected failures (e.g., network time-outs) interrupt replay testing and are logged as such. Instead, a transaction revert is considered an symptom of a behavioral difference and is logged as part of a successful replay session.

Step (3) $T_{outcome}$. The third test method compares the output-based and storage-based oracles with the actual outcomes captured by T_{replay}. A replay test session is deemed successful only if there are no deviations in output. However, CATANA also registers and logs any change in storage so that the tester can examine the impact of the upgrade more in-depth. Specifically, for each transaction replayed, it builds a log entry with the following information:

- **Transaction Data**: Information about the replayed transaction (e.g., its decoded arguments).
- **Output Changes**: A comparison of the transaction's output value on $\text{USC}(P, L_n)$ and $\text{USC}(P, L_{n+1})$. The output value can be equal to the return value of the transaction (if any) or to an error message in case of revert.
- **Storage Changes**: A list of changes in the contract's storage variables after replaying the transaction on $\text{USC}(P, L_n)$ and $\text{USC}(P, L_{n+1})$. Each change entry includes:
  - *Variable Data*: Key information about the storage variable (e.g., its name and type);
  - *Variable Value*: The value of the variable after replaying the transaction on L_n and L_{n+1};
  - *Storage Slot*: The storage slot of the variable after replaying the transaction on L_n and L_{n+1};
  - *Change Type*: The nature of the change. This can be a variable value change, a variable addition, a variable deletion, or a storage slot change.
- **Status Code**: Summarizes whether the replay testing completed successfully (with or without logged changes), or resulted in error (either on the original or the upgraded contract).

Illustrative Example. To better demonstrate how CATANA operates, we present a faulty upgrade scenario involving the `CToken`[7] smart contract from the Compound Finance lending protocol. In this example (see Listing 1), when a user invokes the `mint` function of the `CErc20` contract (line 30), it calls `mintInternal` (line 23), which in turn calls `mintFresh` (line 3). The `mintFresh` function is responsible for minting new tokens and updating the `totalSupply` state variable, which tracks the total number of tokens in circulation. The original contract version correctly increments `totalSupply` by the `mintTokens` amount (line 14). However, the mutated version includes a bug (line 15) to simulate a faulty upgrade. The injected mutation subtracts 1 from the new `totalSupply` value, causing an underreporting in the total amount of tokens. Listing 2 shows an excerpt of replay testing log produced by CATANA when evaluating the faulty contract. When CATANA replays a transaction that targets the `mint` method (line 2), it records both the changes in the transaction output (line 7) and in the contract's storage (line 11). CATANA was able to detect the mutant in the `CToken` smart contract, although the issue was not immediately noticeable from the transaction's output. In fact, the return statement of the `mint` function (line 32 of Listing 1) represents a high-level success indicator. However, the mutation neither caused a transaction revert nor propagated to the return value of `mint`, which is still equal to 0 (indicating no error) for

[7] CToken address: 0x3363BAe2Fc44dA742Df13CD3ee94b6bB868ea376

```
abstract contract CToken {

      function mintFresh(address minter, uint mintAmount) internal {
        uint allowed = comptroller.mintAllowed(address(this), minter, mintAmount);

        if (allowed != 0) { revert MintComptrollerRejection(allowed);  }
        if (accrualBlockNumber != getBlockNumber()) { revert MintFreshnessCheck(); }

        Exp memory exchangeRate = Exp({mantissa: exchangeRateStoredInternal()});
        uint actualMintAmount = doTransferIn(minter, mintAmount);

        uint mintTokens = div_(actualMintAmount, exchangeRate);

--      totalSupply = totalSupply + mintTokens; // Original Version
++      totalSupply = totalSupply + mintTokens - 1;  // Mutated Version (faulty)

        accountTokens[minter] = accountTokens[minter] + mintTokens;

        emit Mint(minter, actualMintAmount, mintTokens);
        emit Transfer(address(this), minter, mintTokens);
    }

     function mintInternal(uint mintAmount) internal nonReentrant {
        accrueInterest();
        mintFresh(msg.sender, mintAmount);
    }
}

contract CErc20 is CToken {
      function mint(uint mintAmount) override external returns (uint) {
         mintInternal(mintAmount);
         return NO_ERROR;
      }
}
```

Listing 1. Differences in the CToken smart contract due to a possible faulty upgrade

```
{
    "tx": {
      "hash": "0x14e6...d2b0",
      "functionName": "mint(uint256_mintedAmount)",
      "input": [{"mintAmount": "1813800000000000000000"}]
    },
    "outputChanges": {
      "valueBefore": "0",
      "valueAfter": "0"
    },
    "storageChanges": [
      {
        "changeType": "value-changed",
        "contract": "CErc20Delegate",
        "name": "totalSupply",
        "type": "t_uint256",
        "slotBefore": "13",
        "slotAfter": "13",
        "decodedValueBefore": 4509423182276372383,
        "decodedValueAfter": 4509423182276372382
      }
    ]
  }
}
```

Listing 2. Excerpt of Catana log for the faulty CToken smart contract

both the original and the mutated contract (line 7 of Listing 2). Still, variables like `totalSupply` are critical to the contract's functionality, and any inaccuracy in its calculation can propagate and affect functions that rely on its value. As shown in the replay testing log, the `storageChanges` section (line 11) uncovers the discrepancy in the `totalSupply` value before and after the upgrade. This helps to visualize the effects of code changes that might not produce immediately obvious symptoms.

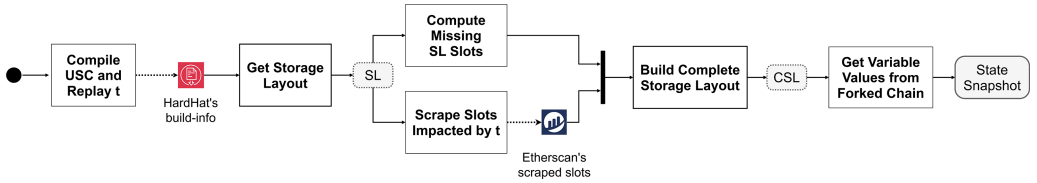


Fig. 5. Storage analysis workflow implemented by Catana.

3.2 Storage Analysis Workflow

During replay testing, Catana analyzes the storage of a proxy contract to uncover changes that may surface when the connected logic contract is upgraded. To understand how state variables are extracted and analyzed, we first refer to Ethereum's *storage model*, introduced in Section 2. Under this model, each smart contract maintains its storage as a *slot-value* database of state variables. Whenever Catana replays a transaction, it must retrieve each variable's data from its corresponding *slot* to reconstruct a complete snapshot of the proxy's state. The Solidity compiler and existing testing utilities (e.g., `hardhat-storage-layout`) produce an SL describing the starting slot of each variable in storage. However, this layout provides an incomplete mapping for complex, nested, or dynamic types (e.g., arrays and structs), whose data can span multiple storage slots. If a replay test involves storage modifications to such data, eventual changes before and after the upgrade would go unnoticed. Catana addresses this limitation by building a **Complete Storage Layout (CSL)**, which maps every component of each variable to its specific location in storage. This richer view of the contract's internal state improves observability during testing and allows developers to know exactly where changes occur after an upgrade. The storage analysis workflow implemented by Catana is depicted in Figure 5, and in the following we describe each step in detail.

(1) Compile USC and Replay t. As part of its replay testing workflow, Catana replays a transaction t on the the target USC (P, L_n) so that its effects are committed to the proxy's storage. As explained in Section 3, this process happens twice, on the original and on the upgraded versions of the USC.

(2) Get Storage Layout. After compilation, Catana extracts the **SL** of the target USC (P, L_n) from HardHat's `build-info` files. The SL describes how the state variables of the smart contract are laid out in long-term memory. Each variable's "identifier" consists of two components: the *starting slot* `p`, indicating where the variable resides, and the byte *offset*, indicating where the variable starts within that slot.

(3) Compute Missing Slots. While the SL provides the starting storage slot `p` for each state variable, this information alone is insufficient for extracting the value of certain variable types. Dynamic, complex, or long data types, such as *arrays*, *structs*, and long *bytes*, require additional steps to locate where their data is stored. For example, let us consider the dynamic array shown in Figure 6. The position of the array in storage is slot `p` = 508, however, the data at this location only describes the array's length. The actual array data starts at `keccak256(p)` and is laid out one element after the other. To manage this and other exceptions, Catana extends the initial SL by systematically computing the storage slots where the data of each state variable can be found, following the rules specified in the Solidity documentation.

(4) Scrape Slots Impacted by t. An additional consideration must be made for the *mapping* data type. Mappings in Solidity are used to store data in the form of key-value pairs and, unlike other data types, knowing the initial slot `p` of the mapping variable is not enough to locate its elements.

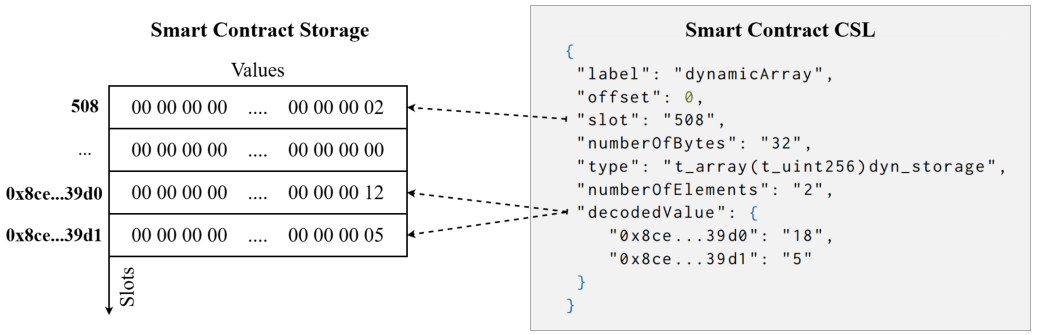


Fig. 6. Example of CSL with decoded variable values.

In fact, as per the Solidity documentation, the value of a specific mapping element is found by applying a hashing function to its key. However, Solidity does not natively store iterable keys due to efficiency constraints in its resource-limited environment. So, if a contract does not explicitly provide the list of mapping keys (e.g., by storing them in a separate array), it is impossible to know them a-priori. One possible solution is to use HardHat's `debug_traceTransaction`.[8] This method returns a detailed trace of a mined transaction, providing a step-by-step breakdown of each executed opcode and the corresponding state of the EVM. By analyzing the opcodes, it is possible to identify the keys of mapping values affected by any t to be replayed. The drawback is that retrieving a single trace can take several minutes and relying on this method would make each replay test extremely inefficient. Instead, we implemented a web scraper that, given a transaction to be replayed t, extracts all the storage slots that were historically impacted by its execution. This information can be retrieved from *Etherscan*'s state change page[9] for any given transaction.

(5) Build Complete Storage Layout. Since Etherscan only provides raw storage slots, without the associated variable names or types, Catana identifies those belonging to mappings by checking if they are already present in the previously computed ones. After this step, the CSL is ready.

(6) Get Variable Values from Forked Chain. After computing the CSL, Catana extracts the value of each variable from storage. The values are retrieved by calling `getStorageAt(address, slot)`[10] which, given the proxy address and a storage slot, extracts the respective value from the active fork. Once the CSL is expanded with the variable's values, Catana decodes them using the respective variable type so that they are in a human-readable format. An example of CSL with variable values for a dynamic array is shown in Figure 6. The output of this process permits Catana to compare the state snapshot for any two versions of an USC (P, L_n) as described in Section 3.1.

4 Research Questions

In the following, we report the key RQs that motivate this work. For each of them we briefly report the intent of the question and the strategy we planned to conduct the experimental studies.

RQ1: How does storage data analysis impact the effectiveness of Capture-Replay testing for USCs? RQ1 investigate to what extent incorporating smart contract storage analysis enhances

[8]HardHat's debug_traceTransaction: https://hardhat.org/hardhat-network/docs/overview#the--debug_tracetransaction--method

[9]Etherscan's state changes page for a transaction: https://etherscan.io/tx/0x6ac79eda39a2d00ab9b4d3440c4672aa0493f99f89f265d8734b3a8aa9f28d14#statechange

[10]HardHat's getStorageAt: https://hardhat.org/hardhat-network-helpers/docs/reference#getstorageat(address,-index,-[block])

Catana's ability to detect disruptive upgrades. Smart contracts rely heavily on state variables for their operations, and an upgrade may alter how the contract interacts with its storage, potentially affecting the outcomes of future transactions. By analyzing state changes before and after an upgrade, we can gain additional insights into how modifications might influence the behavior of the USC. This is especially important in cases where subtle changes do not immediately manifest through transaction outputs (for example, through a revert). To answer this question, we simulate disruptive upgrades by generating mutants of USCs deployed on the Ethereum Mainnet. We then evaluate mutant detection by running a comprehensive replay test suite built from the USC's full transaction history. To measure the impact of storage analysis, we check how many faulty upgrades can exclusively be detected through changes in smart contract storage.

RQ2: Which transaction selection policy most effectively identifies disruptive behaviors in USCs? Replay testing is a powerful technique, but using the entire transaction history in real-world scenarios can be prohibitively expensive. Executing a replay test on a USC can take a few seconds due to overhead such as retrieving past blockchain states and configuring the local environment. Although this cost is acceptable when testing a single USC, it is still important to select transactions in a way that balances high fault-detection effectiveness against efficiency. Therefore, RQ2 investigates four distinct policies (*random*, *last*, *frequency*, and *unique*) for replay test selection. The study refers to several iterations, each one increasing the maximum size of the test suite (i.e., the maximum number of tests, or transactions to be replayed). We gauge each policy's performance primarily by the *Mutation Score*, which indicates how effectively a replay test suite identifies potentially disruptive changes. However, we also track the number of transaction replays (i.e., test executions) needed to kill the injected mutants so as to measure testing overhead. This permits us to assess the practical tradeoffs of replay testing across different selection policies and test suite sizes.

5 Experiment Methodology and Setup

In order to answer the RQs presented in Section 4, we need a set of Ethereum USCs that were upgraded since deployment. Although the proxy pattern is often embedded in real-world applications, developers approach upgrades with caution due to the connected risks and overall lack of supporting frameworks for regression testing. Thus, in practice, the underlying logic is replaced only infrequently [28, 34]. Since each USC typically features one or two past implementation versions, conducting large-scale experiments would require retrieving and testing hundreds of projects to capture a sufficient number of upgrade instances. To address this limitation, we simulate disruptive upgrades by generating mutants from the logic contract L of selected USCs. The rest of this section is organized as follow: Section 5.1 describes the experimental subjects, including the selection criteria for USCs and their characteristics. Sections 5.2 and 5.3 present the methodology and the evaluation metrics for RQ1 and RQ2. Lastly, Section 5.4 describes the hardware setup used for the experiment.

5.1 Subjects

To select the study subjects, we defined the following criteria:

(1) The subject must be a proxy-based USC deployed on the Ethereum Mainnet. This allows us to capture transactions from real end-users.
(2) The source code of the USC must be verified on Etherscan. Access to the ABI and source code is essential for interacting with the contracts during testing and for extracting their SL.
(3) The USC must be written in Solidity and use version 0.5.13 or later. This criterion is set because prior versions of the Solidity[11] compiler did not produce the `storageLayout` output.

[11]Solidity 0.5.13 release announcement: https://soliditylang.org/blog/2019/11/14/solidity-0.5.13-release-announcement/

Table 1. Experimental Subjects

ID	Name	Contract Addresses	StartBlock	EndBlock	#T	u/#ABI
1	BSTPStaking	P: 0x57ba886442d248c2e7a3a5826f2b183a22ecc73e L: 0x5d2c0cc239b33ffc01337c90194acacd50c79088	16483398	20357000	1175	4/18
2	Paladin	P: 0x241326339ced11ecc7ca07e4aa350234c57f53e5 L: 0xcf131548b18d55fb29df2df47b360c41389ebb2b	14880569	20357000	578	3/20
3	GMMToken	P: 0x4B19C70Da4c6fA4bAa0660825e889d2F7eaBc279 L: 0x60a1B168CE980Ef69250362a66e40f8A7050Ce2F	0	20357000	2547	4/17
4	Lucids	P: 0x7DeF9c6b08718Cc9e326CcB057B6C384e6788AD0 L: 0xea690f45047F6be5E513D35b44933999866C5aA6	0	20357000	1526	10/29
5	DeDudes	P: 0xC28b0F274c6eD418d85f9d9CF77c245ada6091DD L: 0x8dfd8220976a0445b1779e5e9d6cb0bfe7b5dadc	0	20357000	2368	13 /31

(4) The USC must feature up to 3000 successful transactions. To capture all relevant user behaviors, we plan to use the complete transaction history of a USC as a replay test suite. Setting this upper bound keeps the experiment computationally manageable. In fact, one replay test requires approximately 10 seconds[12] to run, and executing thousands of transactions on hundreds of mutants can require many days of computation per project.
(5) The USC must feature more than 500 successfully executed transactions. Projects with fewer transactions may not provide enough data to draw reliable conclusions.

To find viable projects, we used the smart contract search feature provided by Etherscan.[13] We used the "*Proxy*" keyword and sorted the results by latest to prioritize recent Solidity versions (the query was conducted in July 2024). Among the collected results, we selected five projects respecting all criteria. Although this final selection is limited, each USC is mutated to simulate numerous faulty upgrades (more details about this are provided in Section 5.2). This design keeps the experiment's total runtime manageable while still providing comprehensive insight into the capabilities of replay testing. Table 1 lists the subjects selected for the experiment, including the address of the proxy P, the address of its current logic contract L, and the total number of transactions *#T* that were successfully executed on it at the time of writing (i.e., until *EndBlock* = 20357000). If the subject had never been upgraded since deployment, we extracted all the transactions executed on *P* from block 0 to *EndBlock*. Otherwise, we identified the block *b* including the upgrade transaction for *L*, and we set *StartBlock* = *b* + 1. This ensures that all the extracted transactions target the latest logic contract version *L*. Regarding the transaction history, Table 1 also shows the total number of unique methods *u* that were invoked at least once, out of all methods belonging to the USC's interface (i.e., its ABI). Note that we excluded *pure* and *view* functions from the *u*/*#ABI* ratio, even if these are technically part of a smart contract's interface. Indeed, these are read-only functions that cannot change the state of the smart contract, meaning their execution does not produce recorded transactions.

5.2 Methodology for RQ1

RQ1 examines the impact of smart contract storage analysis on the effectiveness of capture-replay testing. To answer this question in a comprehensive way, we observed how disruptive upgrades (simulated through mutants) are detected when using the full transaction history as a replay test suite. Below we detail the steps taken to answer RQ1:

(1) **Replay Test Selection:** For each USC (P, L), we retrieved the set of all successfully executed transactions T (see Table 1) recorded on the Ethereum Mainnet. We use these to

[12]Based on prior empirical experience
[13]Etherscan's smart contract search feature: https://etherscan.io/searchcontractlist

build a *full* replay test suite, setting an upper bound of possible interactions for disclosing mutants.

(2) **Mutant Generation:** We generated a set of mutants for the logic contract L, excluding interfaces, utilities, and linked libraries. Each mutant introduces a single fault into the original contract code. We generated the mutants using SuMo [5], a mutation testing tool for the Solidity language. Specifically, we applied all the mutation operators (both traditional and Solidity-specific) presented in the most recent literature [3]. After mutant generation, we applied random sampling to select a manageable subset of 100 mutants.

(3) **Replay Testing:** We ran CATANA on each (compilable) mutant $m \in M$ using the *full* replay test suite. This process produces a *replay matrix* detailing the outcome of each transaction $t \in T$ on each mutant m.

(4) **Killed Mutants Analysis:** From the replay matrix, we identified the set of *killed* mutants $K_{\text{full}} \subseteq M$. A mutant m is considered *killed* if at least one transaction in the *full* replay test suite could detect it using either of the following killing conditions:
   - **Output Killing Condition:** There exists at least one transaction $t \in T$ for which the output produced by the original contract differs from that produced by mutant m. This includes any discrepancy in the return value or in the revert behavior of the transaction (e.g., the transaction is successfully executed on the original contract but it reverts on the mutant).
   - **Storage Killing Condition:** There exists at least one transaction $t \in T$ that results in a change in the contract's storage state when executed on the original contract compared to mutant m. This includes scenarios where at least one state variable in m is added, deleted, altered in value, or moved to a different storage slot.

To answer RQ1, we examine the outcome of CATANA on the killed mutants K_{full}, categorizing them as follows:

- $K_o \subseteq K_{\text{full}}$: is the subset of mutants killed exclusively by the **output killing condition**, indicating detectable functional differences in the contract's response[14];
- $K_s \subseteq K_{\text{full}}$: is the subset of mutants killed exclusively by the **storage killing condition**[15];
- $K_{os} \subseteq K_{\text{full}}$: is the subset of mutants killed by both the **output** and the **storage killing condition**.

To consider the storage-based replay testing approach as effective, the mutants in K_s should represent a meaningful contribution to K_{full}. A significant K_s implies that storage data extraction can enhance Capture-Replay testing for USCs, uncovering unique issues that output-based conditions might miss.

5.3 Methodology for RQ2

RQ2 explores different policies for selecting transactions from the complete history and evaluates how the generated test suites perform in detecting disruptive upgrades. To this end, we compared their effectiveness and efficiency in killing USC mutants. Below we detail the steps taken to answer RQ2:

(1) **Replay Test Selection:** To build the replay tests suites to be evaluated, we applied several selection policies to the full transaction history T of each target USC (P, L). The selection policies are:

[14]In Mutation Testing these mutants are said killed under Strong Mutations [31]
[15]In Mutation Testing these mutants are said killed under Weak Mutations [31]

— **Random:** This policy randomly selects n_R transactions from T, providing broad coverage without bias toward specific blocks or methods.
— **Last:** This policy selects the last n_L transactions in T based on their block number, focusings on recent user behaviors that may reflect the latest usage patterns.
— **Frequency:** This policy selects n_F transactions so that the sample mirrors the original method distribution in T. This guarantees that the resulting test suite includes transactions for those methods that are most frequently invoked in real-world usage.

For each project, n determines the maximum **size of the replay test suite** built using the different policies (i.e, $n = n_R = n_L = n_F = n_U$). This is computed as $n = u \times i$, where u represents the total number of unique method signatures in the transaction history (as shown in Table 1), and i is the scaling factor. In the experiment, we incrementally increased the value for i to observe the effect of different test suite sizes on mutant detection.

(2) **Extraction of Killable Mutants:** Because some contract statements in L may not be invoked by any transaction in T, certain mutants in M might be inherently undetectable. Such mutants would not provide meaningful insight to this comparative study, so the set of *killable* mutants for RQ2 is given by $K_{\text{full}} \subseteq M$. Indeed, these are the mutants that were *actually* killed when using the full transaction history in RQ1 as a replay test suite (regardless of the considered killing condition).
(3) **Replay Testing:** To evaluate each policy's effectiveness, we run the generated replay test suites against the mutants in K_{full}. Since we must run replay testing for varying test suite sizes and multiple repetitions, this process is computationally expensive. For each policy, we repeat the selection and the replay testing process 10 times. The only exception is the policy *last* which is fully deterministic and not influenced by randomness.

To answer RQ2, we compare the defined transaction selection policies considering the following metrics:

(1) *MS*: The **Mutation Score** (*MS*) represents the proportion of killed mutants $K \subseteq K_{\text{full}}$ detected by a replay test suite. A higher *MS* estimates better effectiveness in detecting potential issues in the USC Under Test.
(2) *Replays*: The number of **transaction replays** (i.e., test executions) necessary to achieve a certain *MS* value. This metric represents the efficiency of the test suite in terms of transaction usage. Fewer replayed transactions indicate a more efficient and targeted approach.

5.4 Hardware Setup

We conducted the experiments by running Catana on Ubuntu Linux Virtual Machines with 16 GB RAM and 1vCPU. For the extraction of historical blockchain data, Catana supports forking of the main network through Hardhat.[16] To use this feature, Hardhat must connect to an archive node, which is an instance of an Ethereum client configured to hold historical blockchain states. For typical usage of Catana having a personal archive node is not necessary, as users can rely on free services such as Alchemy or Infura. However, access through these services is mostly limited and not suitable for intensive use. Since our experiments involve replaying thousands of transaction executions on numerous mutants, we set up our own archive node using Erigon,[17] an execution layer client designed to offer an optimized implementation of Ethereum. This allows us to handle the extensive data requirements of our experiments while maintaining consistent performance.

[16]Hardhat's forking feature: https://hardhat.org/hardhat-network/docs/guides/forking-other-networks
[17]Erigon: https://erigon.tech/

Table 2. List of Mutants Randomly Sampled from Those Generated by SuMo's Mutation Operators

Mutation Operator	Illustrative example of a possible mutation	Mutants
ACM - Arg. Change of overloaded Method call	`foo(a,b) ⟶ foo(a,b,c)`	4
AMR - Array Member Replacement	`a.push(x) ⟶ //a.push(x)`	1
AOR - Assignment Operator Replacement	`balances[to] += 1 ⟶ balances[to] = 1`	5
AVR - Address Value Replacement	`foo(address_a) ⟶ foo(address_b)`	33
BOR - Binary Operator Replacement	`a + b ⟶ a - b`	169
BVR - Boolean Value Replacement	`true ⟶ false`	20
CER - Casting Expression Replacement	`uint64(a) ⟶ uint56(a)`	3
CSC - Conditional Statement Change	`if(condition) ⟶ if(true)`	30
EHD - Exception Handling Deletion	`revert() ⟶ //revert()`	30
FCD - Function Call Deletion	`foo() ⟶ //foo()`	16
GVR - Global Variable Replacement	`msg.value ⟶ msg.value-1`	5
ISD - Initialization Statement Deletion	`initialize(){ setFee(1); } ⟶ initialize(){}`	4
LSC - Loop Statement Change	`while(condition) ⟶ while(false)`	2
PLR - Precision Loss Replacement	`(a * b) / c ⟶ (a / c) * b`	8
MCR - Math-Crypto function Replacement	`keccak256 ⟶ sha256`	10
MOD - Modifier Deletion	`function foo() onlyOwner {} ⟶ function foo() {}`	25
NVR - Number Value Replacement	`1000 ⟶ 1001`	95
OLFD - Overloaded Function Deletion	`function foo(){} ⟶ /*function foo(){}*/`	10
ORFD - Overridden Function Deletion	`function foo() override {} ⟶ /* function foo() override {}*/`	11
RSD - Return Statement Deletion	`return fee ⟶ //return fee`	10
RVS - Return Values Swap	`foo() returns (uint a, uint b) ⟶ foo() returns (uint b, uint a)`	1
SKID - Super Keyword Insertion and Deletion	`supply ⟶ super.supply`	3
SVR - String Value Replacement	`''TOKEN'' ⟶ ""`	1
TRC - Transfer Return Check	`require(transferTo(to, amount))) ⟶ transferTo(to, amount)`	2
UORD - Unary Operator Replacement Deletion	`!a ⟶ a`	2
Total		**500**

6 Discussion of Results

In this section, we discuss the results of the experiment and answer the RQs posed in Section 4. The data supporting the findings of this study are publicly available on GitHub.[18] We recall that the objective of RQ1 is to evaluate whether storage analysis can significantly improve Catana's ability to detect disruptive smart contract upgrades. Instead, RQ2 studies the effectiveness of several transaction selection policies in identifying any change exhibited by upgraded USCs (i.e., in the output, in the storage, or in both). To address both these questions, we first used the full transaction history T of each selected USC (P, L) to run replay testing on mutants of their logic contract L. Table 2 summarizes the 500 randomly generated mutants across all projects. It outlines the corresponding mutation operators and provides a code snippet to illustrate how each one alters Solidity contracts. A single mutation operator may embed multiple mutation rules, thus the example is intended for illustrative purposes only. For a more detailed explanation of the operators, please refer to our previous work [3, 5].

Table 3 summarizes the replay testing results for each subject. Here, $\#M$ indicates the total number of mutants that could be compiled and tested, while $\#T$ is the size of the replay test suite (i.e., the total number of successful transactions available on Ethereum). The total number of replay test executions for each project is computed as $\#(M \times T)$. Although generating a complete replay matrix can be time intensive (testing continues even after a mutant is detected early, resulting in more than 800k replays), this process yields a full *mutant-transaction* mapping that we use for subsequent analyses. Within M, the subset K_{full} denotes those mutants that were killed by at least one transaction in T (regardless of which killing condition triggered detection), while L contains the live mutants that remained undetected. The MS for the *full* replay test suite T indicates the upper bound on potential mutant detection for any smaller test selection performed on T. In

[18]Replication package repository https://github.com/MorenaBarboni/Catana_ReplicationPackage

Table 3. Results of Replay Testing Using the Full Transaction History T as a Test Suite

ID	$\#T$	$\#M$	$\#K$	$\#L$	MS	$Replays$	$Time$
1	1175	98	48	50	48.9%	117500	205 h
2	578	95	20	75	21%	57800	85 h
3	2547	92	12	80	13%	254700	509 h
4	1526	93	73	20	78.5%	152600	403 h
5	2368	90	85	5	94.4%	236800	596 h
Tot.	8194	468	238	230	50.9%	819400	1798 h

Legend:
#T: Transactions in the Test Suite
#M: Tested Mutants
#K: Killed Mutants
#L: Live Mutants
MS: Mutation Score = $(\#K/\#M) * 100$
Replays: Number of Replay Test Executions = $\#(T \times M)$

Table 4. Mutants Killed by the Full Transaction History T

ID	$\#M$	$\#K_{\mathbf{full}}$	$\#K_o$	$\#K_s$	$\#K_{\text{os}}$	MS
1	98	48	0	42	6	48.9%
2	95	20	2	7	11	21%
3	92	12	2	6	4	13%
4	93	73	0	70	3	78.5 %
5	90	85	0	81	4	94.4%
Tot.	468	238	4	206	28	50.9%

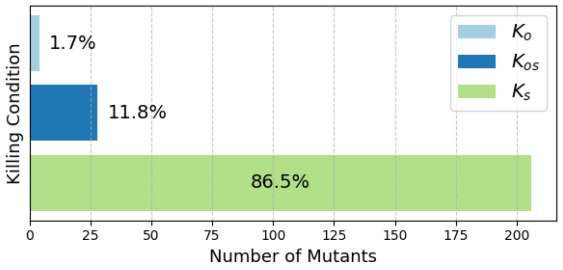


Fig. 7. Distribution of mutants in K_{full} by killing condition

Section 6.1, we focus on the subset of mutants that the full transaction history T was able to kill and analyze the specific factors contributing to their detection. In Section 6.2, we present a comparative study of different transaction selection policies applied to T, examining both their effectiveness and efficiency in detecting the killable mutants K_{full}. Based on this investigation, Section 6.3 outlines insights and guidelines for practitioners aiming to ensure the reliability of their contract upgrades.

6.1 Answering RQ1 - Impact of Storage Analysis

With RQ1, we investigate whether and to what extent the extraction of storage data influences Capture-Replay testing outcomes in USCs. To answer this question, we focus on the subset of mutants K_{full} that were killed when replaying the *full* transaction history T (see Table 3), exploring the reason for their detection. Table 4 further categorizes these mutants by killing condition into $\#K_o$ (mutants detected by a change in *output*), $\#K_s$ (mutants detected by a change in *storage*), and $\#K_{os}$ (mutants detected by *both*). As shown in Figure 7, a replay testing approach that relies solely on examining transaction outputs is highly limited. In fact, only around 13.5% ($K_o \cup K_{\text{os}}$) of the killable mutants across all projects produced observable output differences with respect to the original smart contract. Specifically, 11.8% of the mutants (K_{os}) resulted both in output and storage changes during replay testing. Within this small set of 28 mutants, all of them were detected because a replayed transaction passed on the original contract but reverted on the mutated one. Since the revert prevented the expected storage modifications from taking place, Catana also logged changes in the contract's state variables. Instead, 1.7% of the mutants (K_o) were detected by observing exclusively the output of a transaction. Of these four mutants, one of them was killed by a revert, and the remaining three due to changes in the transaction's return value. The majority of the killable mutants (86.5%) could only be detected by observing changes in the smart contract's storage.

In general, smart contract methods that can potentially change the state (i.e., those not marked as *pure* or *view* in Solidity) do not return values directly to the external caller. While it is technically possible to simulate the execution of such methods and observe their outputs (e.g., via a static call), the return values from the contract's internal execution are often lost or

Table 5. Types of Changes That Triggered Mutant Detection During Replay Testing

Set	Mutants	Tx. Output Changes		State Variable Changes			Total
		Return Value	*Revert*	*Value*	*Slot*	*Value and Slot*	
K_o	4	3	1	-	-	-	4
K_s	206	-	-	242	3	5	250
K_{os}	28	0	28	63	0	0	91

Legend:
K_o: set of mutants exclusively killed by output changes
K_s: set of mutants exclusively killed by storage changes
K_{os}: set of mutants killed by both output and storage changes

sometimes simplified at the ABI level. For example, state-changing functions in the ABI might return higher-level indicators, such as whether the transaction succeeded or failed (as in the illustrative example in Listing 1), rather than more granular data. Therefore, unless a code change causes a transaction to revert, its effects frequently remain hidden within the contract's state variables and are not immediately evident.

As detailed in Table 5, all mutants in K_s collectively caused 250 changes in state variables: 242 of these involved value changes, 3 involved slot changes, and 5 involved both slot and value changes. There is also a notable imbalance in the types of variables typically affected. The vast majority of changed values are elements of a mapping (82%), followed by unsigned integers (14.4%), and a small percentage of address (1.6%), boolean (1.2%), and string (0.8%) data types. Although the distribution of affected variables depends on the specifics of each project, mappings are clearly critical to smart contract functionality. In fact, they are frequently used to store user data, such as token balances, staking amounts, or configuration parameters. Thus, being able to locate and inspect their elements is crucial to fully understand the impact of code changes and prevent potential issues in real-world contracts.

Answer to RQ1: Integrating storage analysis in replay testing is essential for flagging upgrades that might disrupt the behavior of the USC, leading to regressions and compatibility concerns. In our experiment, 86.5% of the killable mutants could exclusively be detected by observing changes in the smart contract's state variables.

6.2 Answering RQ2 - Analysis of the Transaction Selection Policies

The results from RQ1 demonstrated that Capture-Replay testing can effectively detect disruptive changes in USCs. However, using the entire transaction history T as a replay test suite may be unfeasible, especially for projects with hundreds of thousands of historical transactions. Running a single replay test takes on average 7 seconds, mostly because Catana must fork the Ethereum Mainnet and retrieve historical states. While this overhead is still reasonable when testing a single USC (as opposed to a large number of mutants), it becomes crucial to select only the most relevant transactions in order to maximize effectiveness (i.e., fault-detection capabilities) and efficiency (i.e., reduced replay time). Therefore, RQ2 investigates how different transaction selection policies (*random*, *last*, *frequency*, and *unique*, as defined in Section 5.3) detect disruptive behaviors in USCs. This comparative study evaluates replay test suites of different sizes, progressively allocating a higher budget for the testing activities. We recall that the size n of a test suite is computed as $n = u \times i$, where u is the number of unique method signatures in a project's transaction history, and i is the scaling factor. For context, Table 6 shows the u value for each project, together with the size n of its replay test suite for the smallest ($i = 1$), medium ($i = 10$), and largest ($i = 20$) value of the scaling factor. For example, the BSTPStaking project (with 4 unique method signatures) yields a minimum test suite size of 4 transactions and a maximum of 80 transactions (4×20). The maximum

Table 6. Details about the Smallest, Medium, and Largest Replay Test Suite Generated for Each Project

ID	Project	$\#K_{full}$	u	Test Suite (for i = 1)		Test Suite (for i = 10)		Test Suite (for i = 20)	
				Size n	Max. Replays	Size n	Max. Replays	Size n	Max. Replays
1	BSTPSstaking	48	4	4 tx	192	40 tx	1920	80 tx	3840
2	Paladin	20	3	3 tx	60	30 tx	600	60 tx	1200
3	GMMToken	12	4	4 tx	48	40 tx	480	80 tx	960
4	Lucids	73	10	10 tx	730	100 tx	7300	200 tx	14600
5	DeDudes	85	13	13 tx	1105	130 tx	11050	260 tx	22100

Legend:
- $\#K_{\text{full}}$: number of killable mutants in the project
- **u**: number of unique method signatures in the transaction history
- **i**: scaling factor
- **n**: replay test suite size (computed as $u \times i$)

Table 7. Average Results of Replay Testing on the Mutants (Time in Hours, and % of the Killed Mutants Over K_{full})

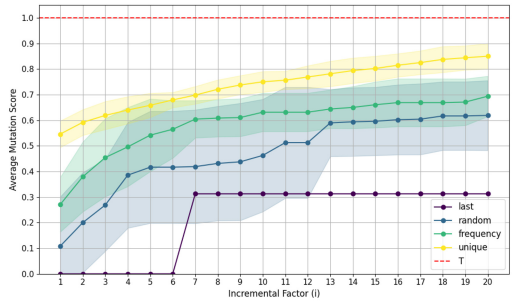
ID	Policies												T
	Last			Random			Frequency			Unique			
	i=1	i=10	i=20	i=1	i=10	i=20	i=1	i=10	i=20	i=1	i=10	i=20	Full History
1	**0.36h**	**3.23h**	**5.64h**	**0.34h**	**2.45h**	**4.00h**	**0.37h**	**2.01h**	**3.29h**	**0.29h**	**1.14h**	**1.69h**	**21.2h**
	0.0%	31.3%	31.3%	10.8%	46.3%	61.9%	27.0%	63.1%	69.4%	54.6%	75.0%	85.0%	100.0%
2	**0.06h**	**0.26h**	**0.40h**	**0.06h**	**0.14h**	**0.20h**	**0.05h**	**0.13h**	**0.17h**	**0.05h**	**0.12h**	**0.17h**	**0.8h**
	70.0%	80.0%	85.0%	78.5%	92.0%	95.5%	76.0%	94.5%	96.0%	89.5%	93.0%	94.0%	100.0%
3	**0.06h**	**0.23h**	**0.39h**	**0.04h**	**0.17h**	**0.31h**	**0.06h**	**0.15h**	**0.23h**	**0.05h**	**0.12h**	**0.17h**	**4.9h**
	75.0%	83.3%	83.3%	85.0%	85.8%	85.8%	87.5%	91.7%	92.5%	91.7%	91.7%	91.7%	100.0%
4	**2.13h**	**13.21h**	**23.10h**	**1.89h**	**15.02h**	**26.23h**	**1.91h**	**13.90h**	**23.43h**	**1.48h**	**5.46h**	**8.60h**	**58.6h**
	1.4%	50.7%	60.3%	12.5%	38.8%	51.2%	16.3%	46.3%	59.32%	31.8%	55.6%	67.0%	100.0%
5	**2.57h**	**17.97h**	**42.99h**	**2.98h**	**18.70h**	**37.76h**	**2.88h**	**15.93h**	**28.49h**	**2.34h**	**7.25s**	**10.94h**	**194h**
	35.3%	37.7%	41.2%	13.5%	43.4%	58.4%	21.9%	59.3%	75.8%	36.9%	67.4%	71.1%	100.0%

number of replays for a given test suite is given by $n \times K_{\text{full}}$. This upper bound is reached in the worst-case scenario when no mutant is detected early.

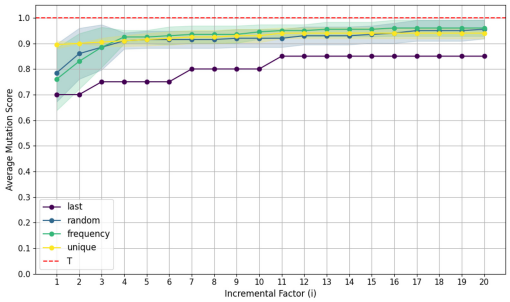
The generated replay test suites are evaluated considering two key metrics. The primary evaluation metric is the **Mutation Score** (MS), as it measures how effectively a replay test suite uncovers changes within upgraded contracts. However, the study also evaluates the efficiency of each test suite by tracking how many **Replays** are needed to kill the injected mutants. Figure 8 illustrates the mutant-detection capabilities of *random*, *last*, *frequency*, and *unique* for each tested project (averaged over 10 repetitions of the experiment). The y-axis shows the MS achieved by the generated test suite, while the x-axis represents the scaling factor i, which increases the test suite size linearly. In each iteration i, we retain all $u(i-1)$ transactions selected in the previous iteration and add u new transactions. Table 7 provides a concrete time benchmark for these policies considering the reference test suite sizes of Table 6. Each cell shows the total time (in *hours*) required for completing replay testing on the mutants in K_{full}, and the corresponding MS. For comparison, the last column refers to the test suite built from the full transaction history T.

Instead, the average performance of each policy on all the considered projects is summarized in Figure 9. Here, the left y axis represents the effectiveness of a policy in terms of its average *MS*, while the right y axis indicates its efficiency in terms of average *Replays*. In the following, we discuss the results by policy in more detail.

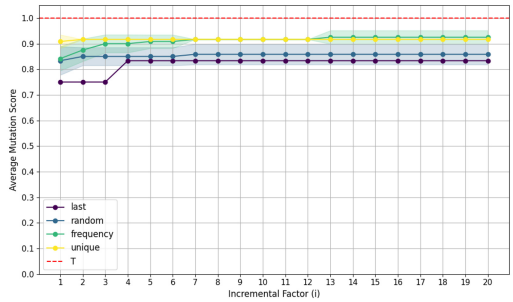
Last Policy. The *last* policy selects the most recent n transactions from the transaction history, focusing on the ABI methods that users have recently invoked. In this case, the test suite size n directly impacts the length of the temporal window from which the transactions are selected. However, since the content of this window is dependent on the behavior of users, the resulting test suite might lack diversity in the methods it covers. Therefore, it often struggles to provide comprehensive functional coverage and detect a broad range of mutants. From the charts in



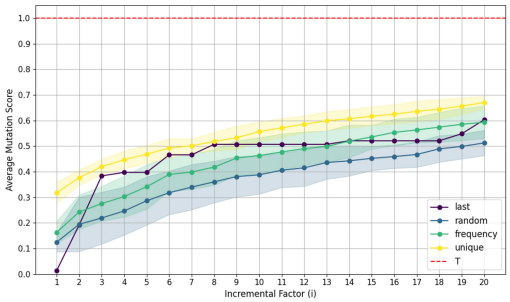
(a) Mutation Score for BSTPStaking over 48 Killable Mutants



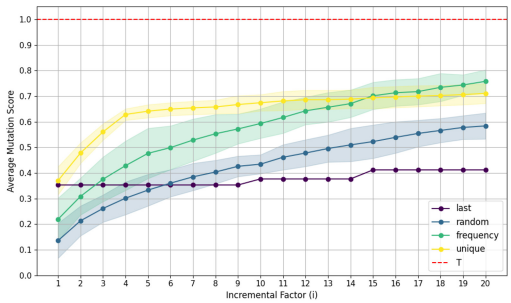
(b) Mutation Score for Paladin over 20 Killable Mutants



(c) Mutation Score for GMMToken over 12 Killable Mutants



(d) Mutation Score for Lucids over 73 Killable Mutants



(e) Mutation Score for DeDudes over 85 Killable Mutants

Fig. 8. Average MS achieved by each transaction selection policy for increasing test budgets.

Figure 8, *last* consistently achieves the lowest MS of all policies. This gap is particularly evident for projects like BSTPStaking (Figure 8(a)), Paladin (Figure 8(b)), and GMMToken (Figure 8(c)), where users engage with only a limited number of methods in the ABI (resulting in very low u values). Figure 9 further shows that, at its smallest test suite ($i = 1$, $\overline{n} = 7$ transactions), *last* achieves an average $\overline{MS} = 0.36$ with 367 test executions, which is only slightly below the *random* policy. However, due to the strong dependence on user-specific behavior, *last* becomes increasingly ineffective and unpredictable as i grows. In fact, selecting redundant transactions inflates the overall number of test executions without proportionate gains in MS. For instance, at the largest test suite tested ($i = 20$, $\overline{n} = 136$ transactions), *last* only reached $\overline{MS} = 0.6$ at the cost of more than 5300 replay test executions. Although it is not especially effective as a stand-alone policy, *last* can be a valuable addition to other strategies by capturing newly emerging user behaviors that might otherwise be overlooked.

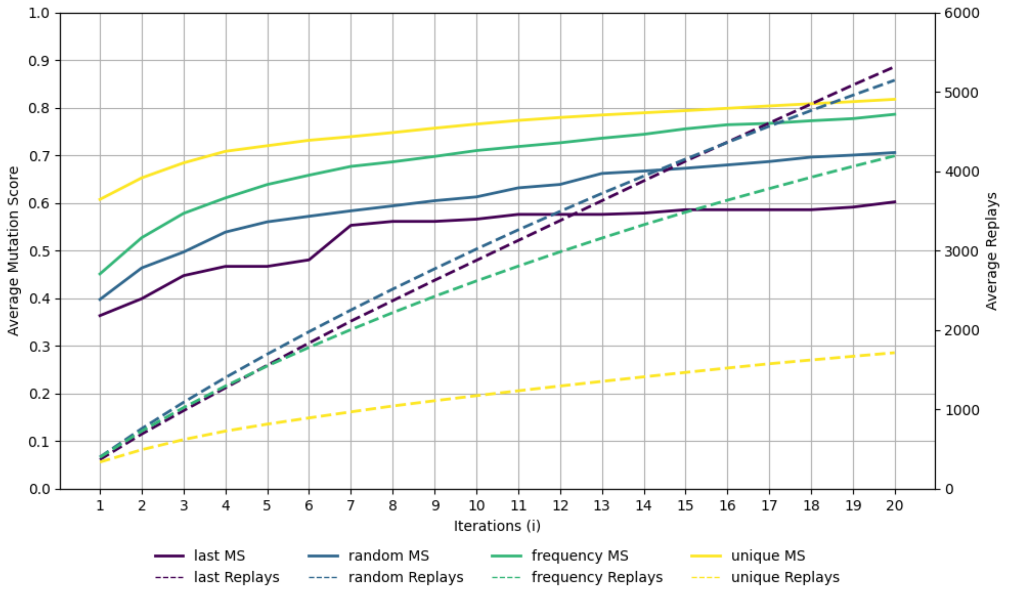


Fig. 9. Average MS and transaction replays achieved across policies.

Random Policy. The *random* policy serves as a baseline approach, offering broad sampling without focusing on specific methods or block numbers. As shown in Figure 8, test suites with randomly selected transactions perform moderately well across all projects, but their performances vary significantly across multiple experiment repetitions. Allocating a larger test budget pays off in terms of mutant detection, with the MS steadily increasing as more transactions are incorporated into the replay test suite. Despite this gradual improvement, for any given value of i, *random* is always outperformed by more focused policies like *frequency* and *unique*. When considering its average performances over all projects, Figure 9 shows that *random* achieves a $\overline{MS} = 0.4$ at the lowest value of i (where $\overline{n} = 7$ transactions). This places it just above the *last* policy, ($\overline{MS} = 0.36$), but still below the *frequency* policy ($\overline{MS} = 0.45$) for roughly the same number of replays. As the test suite size grows, this gap in fault-detection becomes more evident. With the largest test suite we evaluated ($\overline{n} = 136$ transactions), *random* reaches a moderate $\overline{MS} = 0.71$ with around 5100 test executions. With the same test suite size, *frequency* not only achieves a higher $\overline{MS} = 0.79$, but also detects the injected changes faster, with $\approx$ 1000 fewer replays. Therefore, although *random* can eventually achieve adequate MS values, it tends to require more replays and remains inherently more unpredictable in its results.

Frequency Policy. The *frequency*-based selection ensures that heavily used methods are proportionally well-represented in the test suite. This approach prioritizes testing core contract methods that are frequently invoked, but it may neglect less common functions in rarely explored code paths. Like for the *random* policy, it is evident that the selected transactions for small values of i may not extend beyond those commonly invoked by the users. Once the size of the test suite grows, *frequency* systematically broadens the diversity of test inputs for frequent transactions while also incorporating replay tests for less common methods. This tends to drive up MS faster and more reliably than random sampling alone. This trend is evident in Figure 9, where the average MS across projects grows from an initial 0.45 (for $i = 1$, $\overline{n} = 7$ transactions) to almost 0.65 (for $i = 5$, $\overline{n} = 35$ transactions) in the span of five iterations. Once $i = 20$, *frequency* achieves

a maximum $\overline{MS} = 0.79$ transactions, and displays the highest fault-detection capabilities on the Paladin (Figure 8(b)), GMMToken (Figure 8(c)), and DeDudes (Figure 8(e)) projects. Overall, *frequency* performs reasonably well even with modest test suite sizes. When balanced properly, it can offer strong fault-detection capabilities and cost-effectiveness, making it a suitable choice for most projects.

Unique Policy. The *unique* policy evenly distributes transaction selections across all methods invoked in a contract's transaction history. Unlike *frequency*, it prioritizes test diversity over repeatedly targeting high-frequency transactions. This approach ensures that every method in the history is tested in at least one configuration, even at the lowest test budget. Remarkably, *unique* achieves relatively high MSs even with a minimal test suite consisting of one transaction per invoked method. For instance, in projects like BSTPStaking (Figure 8(a)), Paladin (Figure 8(b)), and GMMToken (Figure 8(c)), the MS reached by *unique* at its lowest test suite size ($i = 1$) surpasses that achieved by *last* at its highest ($i = 20$). When considering the average performance over all projects (Figure 9), *unique* outperforms every other policy at the lowest i, achieving $\overline{MS} = 0.6$ against the $\overline{MS} = 0.45$ of the closest competitor (i.e., *frequency*). Then, as the i factor increases, the test suite expands with transactions that explore a wider variety of initial contract states and parameters (i.e., test inputs). As a result, it achieves the highest MS of all policies on two out of five projects (BSTPStaking in Figure 8(a), and Lucids in Figure 8(d)), and only slightly below *frequency* on the remaining three projects. When considering the largest test suite size ($i = 20$, $\overline{n} = 136$ transactions), *unique* achieves the highest MS of all policies ($\overline{MS} = 0.82$). Although this value is only slightly ahead of *frequency*'s ($\overline{MS} = 0.79$), *unique* reaches it with approximately 60% fewer transaction replays. Indeed, *frequency* may require a significantly higher number of replays due to over-representation of popular methods, increasing the likelihood of redundant tests. Additionally, *unique* displays a narrow confidence interval, indicating that it is not only effective but also stable and reliable, with minimal variability in mutant-detection performance.

Answer to RQ2: The *unique* policy, which focuses on method-level diversity, outperforms other strategies in terms of mutation adequacy and number of replayed transactions. In the experiment, we observed that even the minimal replay test suite (containing one transaction per each invoked method) achieves good fault-detection capabilities, killing 60% of detectable mutants.

6.3 Guidelines for Practitioners

Based on our investigation of Capture-Replay testing for USCs, we outline the following insights and guidelines for practitioners aiming to ensure the reliability of their contract upgrades.

First, our results in Section 6.1 show that focusing solely on transaction outputs detects a limited subset ($\approx$ 13.5) of potentially harmful changes. Many smart contract functions, especially state-changing ones (i.e., neither pure nor view), do not return meaningful values at the ABI level. Therefore, a revert might be the only outward signal of a problematic upgrade. When using Capture-Replay testing, practitioners should **carefully examine internal state changes** (e.g., storage variables) and not rely exclusively on transaction outputs. Catana ensures that developers gain visibility into critical data structures, such as mappings that hold user balances and other key application data. Instead of guessing where an upgrade might break the logic, developers can get precise feedback on which specific variables diverged from their expected values.

As shown in Section 6.2, adopting the right transaction selection policy is essential in balancing efficiency (minimizing time) and effectiveness (maintaining high fault detection). The optimal test

selection depends on the project's needs and testing budget (i.e., how many transactions should be included in the replay test suite). Our findings suggest that small, method-diverse replay test suites can still provide fast feedback with reasonable effectiveness. Employing specific policies allows Catana to support both quick smoke tests (e.g., *unique* with small test suites) and thorough regression testing (e.g., *frequency* or *unique* with larger budgets) for high-stakes upgrades.

For example, when using the *unique* policy on only one transaction per invoked method (as retrieved from the project's on-chain history), Catana can quickly verify whether the upgraded contract's core functionality remains intact. Because *unique* ensures that every method is tested at least once, even the smallest test suite can detect disruptive changes in rarely executed paths without devoting extensive time to testing. This approach enables a lightweight validation, with a test suite completing in under a minute on the average project. However, it is important to notice that in this study a mutation operator is blindly applied to any portion of a USC matching its mutation rules. Thus, any line of the USC is potentially subject to a fault. If practitioners have the possibility to foresee that their changes will affect only a specific portion of a USC, then they should leverage this information and properly reconsider the tradeoffs on *unique* that we reported above.

For more comprehensive regression testing, Catana can expand the range of replayed transactions through policies like *frequency* or by increasing the budget for *unique*. Methods that users invoke most frequently often include critical logic, and testing them more extensively (i.e., using a wider variety of transaction inputs and starting states) helps uncover more subtle contract breakages. The *frequency* policy, in particular, ensures that the core functions of a USC receive proportionate testing focus while still maintaining diversity across less-invoked methods as the test suite grows. However, practitioners should be mindful of diminishing returns when scaling up test suites. Beyond a certain threshold, additional tests may contribute only marginally while increasing execution time, regardless of the policy used. For instance, the *unique* policy raises the MS from 0.6 to approximately 0.72 within the first four expansions of the test suite, while it yields a more modest increase to 0.82 until the twentieth iteration. Lastly, practitioners should be aware that selecting transactions exclusively based on recent user behavior (using the *last* policy) may limit test diversity. In scenarios where end-users interact with only a small subset of functions, *last* might produce a test suite that neglects certain methods, leaving portions of the contract untested. However, developers who anticipate newly emerging user behaviors can complement *last* with broader selection policies like *unique* or *frequency*. Doing so ensures that new user interactions are not overlooked.

As a final note for practitioners, we remark that the policies discussed in this work only leverage structural information from smart contracts and their historical set of past transactions. This makes our guidelines broadly applicable when planning regression testing for any USC. At the same time, these policies do not take into account domain-specific behaviors or conditions that may influence testing priorities. Therefore, practitioners should consider how characteristics of their specific application might impact the considerations we reported above.

7 Threats to Validity

In this section, we discuss construct, internal, and external factors [37] that may affect the validity of our study.

Construct validity concerns the assumptions and decisions made during the design of the experiment that may influence the final results. One potential threat to construct validity stems from our method for extracting data on mapping variables. Catana identifies mapping elements by scraping Etherscan for storage slots that were historically impacted by replayed transactions. We made this choice because data scraping significantly reduces the time overhead of replay testing compared

to analyzing the debug trace of a transaction. The drawback is that it narrows down the analysis to mapping slots that were accessed by the original transactions. As a result, if an upgrade introduces interactions with new mapping slots that were not previously accessed, Catana may not detect them. Future work should explore alternative approaches to avoid this kind of false negatives while keeping replay testing reasonably fast. Another threat to validity is the risk of non-deterministic replays due to chain-related variables like the block timestamp. Although Catana inherits the forked blockchain's state, simulated transactions may still diverge from real blockchain execution. Catana mitigates this by discarding transactions that fail on the original contract, implicitly filtering out cases highly sensitive to execution context. Additionally, by replaying each transaction on both the original and upgraded contract within the same environment, Catana ensures any discrepancies affect both equally, isolating upgrade-induced differences from external factors.

Additionally, Catana uses Hardhat Network to setup the local blockchain environment in which to replay transactions. Similarly to other blockchain simulators, Hardhat Network allows forking the blockchain state only at the block level, rather than before the specific transaction itself. Therefore, if any two transactions t_1 and t_2 in the same block both target the contract under test, it might not be possible to correctly set-up the starting state for t_2. However, the probability of two related transactions being included in the same block is relatively low. Furthermore, Catana replays any transaction on the original contract to build the test oracle, which helps to mitigate this problem.

Internal validity is the extent to which the observed outcomes in our study can be confidently attributed to the variables under research. A threat to internal validity concerns the implementation of the experiment with respect to the policies *frequency* and *random*. Indeed, maintaining previously selected transactions in subsequent iterations of the experiment implies the selection of a small number of new transactions. In the case of *frequency*, this implementation strengthens the selection of the most frequently invoked methods, to the detriment of the others. The differences between the two policies may otherwise result less evident.

External validity indicates the extent to which the findings of the study can be generalized beyond the specific conditions of the experiment. The primary threat to external validity concerns the number of study subjects, which is limited by the high costs associated with the replay testing experiments. While using the full transaction history allows us to investigate the capabilities of replay testing in a comprehensive manner, the experiment is time-intensive and required us $\approx$ 75 days for all projects. The decision to cap transactions at 3000 per project for computational feasibility may have influenced project selection, favoring newer contracts with lower activity. The insights provided in this work should be further explored on a variety of USCs, as specific characteristics of the considered projects and their usage patterns might influence the results we got.

8 Related Work

This article explored the use of capture-replay testing to validate smart contract upgrades. In this section, we review research closely related to our proposal, focusing on smart contract testing, upgradeability, and Capture-Replay testing.

Smart Contract Testing. Releasing reliable smart contract code is a primary concern for industry professionals working with blockchain [39]. Research in blockchain testing has primarily focused on stand-alone contracts, introducing techniques for defects and vulnerabilities detection, test generation and test adequacy assessment [13]. To date, smart contract developers can rely on a range of static analysis tools (e.g., Oyente [30] and Slither [16]) and fuzzers (e.g., Echidna [20]) to disclose faults and optimize code prior to deployment. Solidity-specific mutation testing approaches were also proposed to improve the fault-detection capabilities of smart contract test

suites (e.g., SuMo [5] and ReSuMo [6]). Several works do not specifically target the on-chain logic of a DApp, but consider the system as a whole. A few solutions build test cases for DApps [18, 42], or detect specific bugs stemming from the bad synchronization between on-chain and off-chain components [44].

Upgradeable Smart Contracts (USC). Despite their flexibility, USCs can introduce several risks. A few empirical studies summarize the characteristics of upgrade mechanisms and discuss their potential impact. Bodell et al. [23] developed a USC taxonomy comprising 11 unique upgradeability patterns and six related security issues. Li et al. [27] investigated the security implications of USC in the wild, covering 60M+ Ethereum smart contracts. Besides characterizing additional design patterns, they described several USC-specifc security issues, such as the lack of restrictive checks within upgradable functions. An empirical study by Liu et al. [28] shows that functionality update and addition are the two dominating intentions in upgrades, but introducing changes could impact impact client-specific compatibility. Indeed, they show that proxy-based contracts frequently encounter ABI-breaking changes, leading to real world broken usages. Qasse et al. [34] explored broader upgradeability concerns, showing that while upgrade patterns are mature and well-documented, their actual adoption is more limited than expected. Although developers aim to implement feature improvements and security fixes, upgrading often leads to unintended consequences for contract reliability and user trust. Moreover, the impact on contract activity levels post-upgrade is a significant concern, as frequent or poorly communicated upgrades may lead to decreased usage.

Capture-Replay Testing for USCs. The capture-replay approach has been widely utilized across various domains to ensure the continued correctness of software as it evolves [2, 21, 24, 26]. For example, in web testing [1, 29], it enables the validation of complex user interactions across different browsers and environments. In mobile testing [7, 17, 19, 35], it allows for the replication of touch gestures, swipes, and other user inputs across various devices. In IoT environments [15], it facilitates the verification of device behavior under diverse network conditions and real-world usage scenarios. In the blockchain domain, the inherent availability of transaction histories offers a unique opportunity for the validation of USCs. However, no framework implements a systematic capture-replay testing approach for proxy-based systems. When considering compatibility issues, most available solutions only focus on the problem of detecting storage collisions. For example, the OpenZeppelin Upgrades plugin [33] implements static security checks to ensure upgrade safety, which does not consider behavioral compatibility. Even if storage remains intact, an upgrade can introduce semantic changes that alter the logic, break assumptions made by existing users or dependent contracts. Instead, transaction replay has been considered in prior works, but none of these address the challenge of ensuring behavioral compatibility in upgradeable contracts. Available dynamic analysis frameworks that leverage historical transactions are designed for different purposes, including: *(1) active monitoring*, *(2) smart contract analysis*, *(3) security analysis,* and *(4) test generation.* In the domain of active monitoring, DappGuard [12] intercepts real-time transactions to classify and learn potential attacks, and to protect smart contracts from them. Other tools use historical transactions for analysis and testing. ContractVis [22] generates test scripts from historical transactions and executes them in a minimal test environment. Unlike Catana, its purpose is to gain insights into the transparency of a smart contract with respect to its dependencies, and expose potential opacity issues. Instead, SmartGift [45] aims to generate tests for newly written smart contracts by extrapolating realistic test inputs from the transactions of other deployed contracts. Lastly, TxSpector [43] and EthScope [41] both provide transaction-level security analysis. TxSpector permits to investigate Ethereum transactions for attack detection. At a high level, it replays history transactions and records EVM bytecode-level traces to detect attacks that exploit

various vulnerabilities (e.g.,: Re-entrancy). EthScope defines a transaction-centric security analytics framework to detect malicious smart contracts on Ethereum. In this case, transaction replay is a means to re-execute suspicious transactions and confirm a malicious behavior.

9 Conclusions and Future Work

This article introduced Catana, a novel replay-based regression testing framework for proxy-based USCs on Ethereum. Catana captures transactions from an already-deployed USC and harnesses them to validate a new logic contract version to be released. When a transaction is replayed on the smart contract under test, Catana checks both its output and its effects on storage, flagging any discrepancy with respect to the deployed USC. In this way, it allows practitioners to carefully examine the behavior of an upgrade under real-world invocations before releasing it on Ethereum mainnet. Catana supports practitioners in three main areas: (1) *detecting disruptive changes* that might indicate regressions or compatibility issues, (2) *validating intentional modifications*, ensuring they align with expectations, and (3) *debugging*, providing detailed pre- and post-upgrade test logs.

To validate Catana, we used the full transaction history of real-world USCs to run replay testing on disruptive upgrades simulated through mutants. Results showed that extracting and analyzing USC storage data significantly enhances replay testing, with the majority of faulty code changes (86.5% of killable mutants) identifiable only through the analysis of state variables. Additionally, our study of four transaction selection policies (*last*, *random*, *frequency*, and *unique*) revealed that strategies emphasizing method-level diversity (*unique*) consistently achieve higher MSs while minimizing the number of replays required. Notably, even small test suites that include just one transaction per invoked method were able to detect up to 60% of disruptive upgrades, offering a reasonably effective and lightweight testing strategy. This finding is particularly relevant for practitioners, as replaying the full transaction history of high-usage contracts is often infeasible.

While our study evaluated replay testing strategies in a domain-agnostic manner, future work could explore application-specific schemata, such as token standards, governance mechanisms, and financial interactions. Developing adaptive selection policies could meet domain-specific requirements and enhance replay testing effectiveness. Additionally, integrating debug trace support in Catana would enable developers to choose between faster execution or more thorough storage analysis based on their specific needs. Lastly, future work could focus on the use of **Large Language Models** (**LLMs**) to improve the replay testing workflow. For instance, LLMs could enhance transaction selection by predicting high-risk areas in the contract based on the introduced changes. Furthermore, LLMs could assist in automatically generating human-readable summaries of detected changes, suggesting potential root causes, or even proposing targeted fixes for identified issues.

References

[1] Silviu Andrica and George Candea. 2011. WaRR: A tool for high-fidelity web application record and replay. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE Compute Society, 403–410.

[2] Majid Babaei and Juergen Dingel. 2021. Efficient replay-based regression testing for distributed reactive systems in the context of model-driven development. In *Proceedings of the 2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 89–100.

[3] Sebastian Banescu, Morena Barboni, Andrea Morichetta, Andrea Polini, and Edward Zulkoski. 2024. Enhanced mutation testing of smart contracts in support of code inspection. In *Proceedings of the 2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 558–566.

[4] Morena Barboni, Guglielmo De Angelis, Andrea Morichetta, and Andrea Polini. 2023. CATANA: Replay testing for the ethereum blockchain. In *Proceedings of the IFIP International Conference on Testing Software and Systems*. Springer, 257–265.

[5] Morena Barboni, Andrea Morichetta, and Andrea Polini. 2022. SuMo: A mutation testing approach and tool for the Ethereum blockchain. *Journal of Systems and Software* 193 (2022), 111445.
[6] Morena Barboni, Andrea Morichetta, Andrea Polini, and Francesco Casoni. 2024. ReSuMo: A regression strategy and tool for mutation testing of solidity smart contracts. *Software Quality Journal* 32, 1 (2024), 225–253. DOI: https://doi.org/10.1007/s11219-023-09637-1
[7] Carlo Bernaschina, Roman Fedorov, Darian Frajberg, and Piero Fraternali. 2017. A framework for regression testing of outdoor mobile applications. In *Proceedings of the 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 179–181.
[8] Antonia Bertolino, Guglielmo De Angelis, and Francesca Lonetti. 2019. Governing regression testing in systems of systems. In *Proceedings of the IEEE ISSRE Workshops*. IEEE, 144–148.
[9] Antonia Bertolino, Guglielmo De Angelis, Breno Miranda, and Paolo Tonella. 2023. In vivo test and rollback of Java applications as they are. *Software Testing, Verification and Reliability*. 33, 7 (2023), e1857. DOI: https://doi.org/10.1002/STVR.1857
[10] Van Cuong Bui, Sheng Wen, Jiangshan Yu, Xin Xia, Mohammad Sayad Haghighi, and Yang Xiang. 2021. Evaluating upgradable smart contract. In *Proceedings of the 2021 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 252–256.
[11] Vitalik Buterin. 2014. A next-generation smart contract and decentralized application platform. *White Paper* 3, 37 (2014), 2–1.
[12] Thomas Cook, Alex Latham, and Jae Hyung Lee. 2017. DappGuard : Active Monitoring and Defense for Solidity Smart Contracts. Retrieved 15 March 2025 from https://courses.csail.mit.edu/6.857/2017/project/23.pdf
[13] Anil Elakas, Hasan Sözer, Ilgin Safak, and Kübra Kalkan. 2024. A systematic mapping on software testing for blockchains. *Cluster Computing* 27, 6 (2024), 7111–7126.
[14] etherscan.io. 2024. Proxy Contract Verification Page — etherscan.io. Retrieved July 2024 from https://etherscan.io/proxyContractChecker
[15] Kaiming Fang and Guanhua Yan. 2020. IoTReplay: Troubleshooting COTS IoT devices with record and replay. In *Proceedings of the 2020 IEEE/ACM Symposium on Edge Computing*. IEEE, 193–205.
[16] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A static analysis framework for smart contracts. In *Proceedings of the 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*. IEEE/ACM, 8–15.
[17] Sidong Feng and Chunyang Chen. 2022. GIFdroid: Automated replay of visual bug reports for android apps. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, 1045–1057.
[18] Jianbo Gao. 2019. Guided, automated testing of blockchain-based decentralized applications. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings*. IEEE/ACM, 138–140.
[19] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. Reran: Timing-and touch-sensitive record and replay for android. In *Proceedings of the 35th International Conference on Software Engineering, ICSE'13*. IEEE Computer Society, San Francisco, CA, USA, 72–81.
[20] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: Effective, usable, and fast fuzzing for smart contracts. In *ISSTA'20: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Virtual Event, USA, 557–560.
[21] Mouna Hammoudi. 2016. Regression testing of web applications using record/replay tools. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. ACM, 1079–1081.
[22] Pieter Hartel and Mark van Staalduinen. 2019. Truffle tests for free–Replaying Ethereum smart contracts for transparency. arXiv:1907.09208. Retrieved from https://arxiv.org/abs/1907.09208
[23] William Edward Bodell III, Sajad Meisami, and Yue Duan. 2023. Proxy hunting: Understanding and characterizing proxy-based upgradeable smart contracts in blockchains. In *Proceedings of the 32nd USENIX Security Symposium, 2023*. USENIX Association, 1829–1846.
[24] Javaria Imtiaz, Muhammad Zohaib Iqbal, and Muhammad Uzair Khan. 2021. An automated model-based approach to repair test suites of evolving web applications. *Journal of Systems and Software* 171 (2021), 110841. DOI: https://doi.org/10.1016/J.JSS.2020.110841
[25] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 654–665.
[26] Ilia Kravets and Dan Tsafrir. 2012. Feasibility of mutable replay for automated regression testing of security updates. In *Proceedings of the Workshop on Runtime Environments, Systems, Layering and Virtualized Environments*. IEEE.
[27] Xiaofan Li, Jin Yang, Jiaqi Chen, Yuzhe Tang, and Xing Gao. 2024. Characterizing ethereum upgradable smart contracts and their security implications. In *Proceedings of the ACM on Web Conference 2024*. ACM, 1847–1858.

[28] Ye Liu, Shuo Li, Xiuheng Wu, Yi Li, Zhiyang Chen, and David Lo. 2024. Demystifying the characteristics for smart contract upgrades. arXiv:2406.05712. Retrieved from https://arxiv.org/abs/2406.05712

[29] Zhenyue Long, Guoquan Wu, Xiaojiang Chen, Wei Chen, and Jun Wei. 2020. WebRR: Self-replay enhanced robust record/replay for web application testing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Virtual Event, USA, 1498–1508.

[30] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 254–269.

[31] Aditya P. Mathur. 2013. *Foundations of Software Testing, 2/e*. Pearson Education India, India.

[32] OpenZeppelin. 2025. Proxies - OpenZeppelin Docs — docs.openzeppelin.com. Retrieved 15 March 2025 from https://docs.openzeppelin.com/contracts/5.x/api/proxy

[33] OpenZeppelin. 2025. Upgrades Plugins - OpenZeppelin Docs — docs.openzeppelin.com. Retrieved 15 March 2025 from https://docs.openzeppelin.com/upgrades-plugins/

[34] Ilham Qasse, Mohammad Hamdaqa, and Björn Þór Jónsson. 2024. Immutable in principle, upgradeable by design: Exploratory study of smart contract upgradeability. arXiv:2407.01493.Retrieved from https://arxiv.org/abs/2407.01493

[35] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. 2016. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 571–582.

[36] Nicola Ruaro, Fabio Gritti, Robert McLaughlin, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. 2024. Not your type! Detecting storage collision vulnerabilities in ethereum smart contracts. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, San Diego, California, USA. Retrieved from https://www.ndss-symposium.org/ndss-paper/not-your-type-detecting-storage-collision-vulnerabilities-in-ethereum-smart-contracts/

[37] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (2009), 131–164.

[38] Mehdi Salehi, Jeremy Clark, and Mohammad Mannan. 2022. Not so immutable: Upgradeability of smart contracts on ethereum. In *Proceedings of the International Conference on Financial Cryptography and Data Security*. Springer, Springer, 539–554.

[39] Zhiyuan Wan, Xin Xia, David Lo, Jiachi Chen, Xiapu Luo, and Xiaohu Yang. 2021. Smart contract security: A practitioners' perspective. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering*. IEEE, 1410–1422.

[40] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151 (2014), 1–32.

[41] Lei Wu, Siwei Wu, Yajin Zhou, Runhuai Li, Zhi Wang, Xiapu Luo, Cong Wang, and Kui Ren. 2020. EthScope: A transaction-centric security analytics framework to detect malicious smart contracts on Ethereum. arXiv:2005.08278. Retrieved from https://arxiv.org/abs/2005.08278

[42] Zhenhao Wu, Jiashuo Zhang, Jianbo Gao, Yue Li, Qingshan Li, Zhi Guan, and Zhong Chen. 2020. Kaya: A testing framework for blockchain-based decentralized applications. In *Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 826–829.

[43] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. 2020. TXSPECTOR: Uncovering attacks in ethereum from transactions. In *Proceedings of the 29th USENIX Security Symposium*. USENIX Association, Virtual, 2775–2792.

[44] Wuqi Zhang, Lili Wei, Shuqing Li, Yepang Liu, and Shing-Chi Cheung. 2021. Ðarcher: Detecting on-chain-off-chain synchronization bugs in decentralized applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 553–565.

[45] Teng Zhou, Kui Liu, Li Li, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. 2021. SmartGift: Learning to generate practical inputs for testing smart contracts. In *Proceedings of the 2021 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 23–34.

Received 24 January 2025; revised 7 April 2025; accepted 15 May 2025