CollisionRepair: First-Aid and Automated Patching for Storage Collision Vulnerabilities in Smart Contracts

Yu Pan*†, Wanjing Han*†, Yue Duan‡, Mu Zhang†
†*University of Utah, United States* ‡*Singapore Management University, Singapore*
†*{yu.pan, wanjing.han, mu.zhang}@utah.edu* ‡*yueduan@smu.edu.sg*

Abstract

Storage collision vulnerabilities, a significant security risk in upgradeable smart contracts, often arise when a user-facing proxy contract and a backend logic contract share storage space. While static analysis techniques can detect such issues, they often over-approximate program states, leading to false positives and requiring developers to manually verify each issue, giving attackers time to exploit any overlooked vulnerabilities. To address this, we propose CollisionRepair, an automated patching technique for mitigating storage collision risks. CollisionRepair monitors storage access sequences between proxy and logic contracts by defining an "ownership" property for storage locations. It then replays historical transactions to recover existing storage ownership, ensuring the patched code aligns with the current state. A gas impact-aware differential analysis is applied to verify the patch, distinguishing genuine behavioral changes from variations caused by gas usage. Our evaluation on 12,526 real-world vulnerable upgradeable contracts shows that CollisionRepair effectively detects and mitigates storage collision attacks without interfering with normal contract operations.

1 Introduction

Smart contracts are specialized programs that operate on blockchains, driving and transforming various key industries such as NFT marketplaces [4, 5] and decentralized finance (DeFi) [3, 6]. Upgradeable smart contracts (USC) [9, 30] offer a standardized method for updating smart contract logic, enabling changes to the original code without the limitations posed by the immutable nature of blockchains. It allows developers to introduce new features and fix bugs without requiring users to move to new contract addresses, making this approach widely adopted among popular smart contracts (e.g., Compound [3] and OpenSea [4]).

While upgradeable contracts provide great flexibility, they also present additional security risks. A notable concern is the *storage collision* vulnerability [7], often found in upgradeable contracts, where a user-facing proxy contract and a backend logic contract often share the same storage space due to the usage of a special instruction `delegatecall` (more details in Section 2.1). This makes it challenging for developers to implement upgradeable contracts correctly [30], leading to a higher risk of storage-related conflicts. Prior research [42] has shown that hackers can easily exploit these storage collision vulnerabilities to manipulate critical access control flags, creating serious security risks.

To address storage collision vulnerabilities in upgradeable smart contracts, researchers have developed static analysis techniques [30, 42] to detect these issues. However, static analysis inherently struggles with precisely modeling program states, often resulting in over-approximations that lead to significant false positives. For example, UscHunt [30] simply models storage collision issues as inconsistencies in the order of storage variables between proxy and logic contracts without verifying if the mismatched variables are indeed accessed.

In contrast, Crush [42] considers storage collision as a type conflict problem, using symbolic execution to check whether multiple accesses to the same storage location involve incompatible data types. Nevertheless, we argue that a type conflict does not necessarily warrant a storage collision. For instance, when an unsigned integer `x` in the proxy contract and a byte array `a[8]` in the logic contract are stored at the same offset, a write to `x` and another to `a[i]` might suggest a potential type conflict from a static analysis perspective. However, in practice, if these operations access non-overlapping memory cells at runtime—e.g., writing to `x` and `a[7]`—they do not result in a storage conflict.

Because static analysis can generate false positives, developers must carefully review reported issues to identify vulnerabilities before implementing patches. This process can be time-consuming, providing a significant window for attackers to exploit unaddressed security flaws. For example, prior to the notorious DAO attack [17], security experts had already identified the reentrancy bug [49] in the DAO's smart contract. Despite attempts to address the bug, an attacker could

*Yu Pan and Wanjing Han contributed equally to this work.

exploit it before a full solution could be deployed, leading to the unauthorized withdrawal of significant assets from the DAO [15].

To bridge this gap, we propose deploying a first-aid solution to mitigate potential storage collision risks once discovered, providing an immediate response while a comprehensive solution is under development. This approach aims to give developers the time to thoroughly investigate a vulnerability and create a robust patch without leaving the system vulnerable to attack in the interim.

Several efforts [25,33,37,40,41,48,55,57] have been made to automatically mitigate smart contract vulnerabilities. Early approaches [33,40], however, have limitations. They require modifications to the Ethereum Virtual Machine (EVM) [40] or access to Solidity source code [33, 37, 48, 55], reducing their practicality. More recent work [25,41,57], which patches EVM bytecode directly, has shown greater promise. However, these state-of-the-art techniques are not suited for addressing storage collision issues due to three key reasons.

First, storage collision arises from *a sequence of data accesses across two contracts*. In contrast, prior work largely addresses *single-point failures*, such as missing checks for integer overflow, lack of access control, absence of reentrant locks, or inadequate exception handling *within a single contract*. Second, storage collision is inherently *state-aware* because the storage state is significantly influenced by past transactions. However, current methods disregard this factor, assuming that patched code can initiate from a fresh state. Third, prior work [41] assesses the behavioral fidelity of patched code under the assumption that it does not introduce new state-changing instructions. However, we found that patch code can affect contract states, as the additional gas consumption may alter the control flow. Thus, the gas impact-unaware patch verification used in the previous work is imprecise.

To address these limitations, we propose CollisionRepair, an automated patching technique to mitigate storage collision vulnerabilities. To monitor storage access sequences between proxy and logic contracts, CollisionRepair instruments their storage read/write operations to record and verify the "*ownership*" of individual storage locations. To ensure the patched code operates with the current storage state, we *replay historical transactions* to recover the established "ownership" of storage cells. Additionally, to verify the fidelity of patched contracts, we perform a *gas impact-aware* differential analysis to precisely distinguish genuine behavioral discrepancies from diverted control flow caused by varying gap consumptions.

The major contribution of this work is a novel stateful "ownership" model with multi-point monitoring, designed to address storage collisions—stateful bugs that existing methods cannot handle. The key challenge lies in extracting sufficient information from low-level bytecode to track state transitions and safely deploy patches that respect historical state changes.

We have implemented CollisionRepair with 3,367 lines

```
contract Proxy {
    uint64[3] fees;              // slot [0x0]
    address[3] feeRecipients;    // slot [0x1]-[0x3]
    [..]
    address public LOGIC;        // slot [ERC-1967]

    enum FeeType {penalty, transaction, sales}

    constructor() {
        feeRecipients = [0x[..], 0x[..], 0x[..]];
        LOGIC = 0x[..];
        LOGIC.initialize();
    }
    function updateFee(FeeType t, uint64 percent) {
        fees[(uint) t] = percent;
    }
    fallback() external {
        LOGIC.delegatecall(msg.data);
    }
}

abstract contract Initializable {
    bool internal initialized;   // slot [0x0]
    [..]
}

contract Logic is Initializable {
    uint256[256] private __gap; // slot [0x1]-[0x100]
    address admin;              // slot [0x101]
    array artworkIDs;           // slot [0x102]
    mapping artworkHolders;     // slot [0x103]

    function initialize() external {
        require(!initialized);
        initialized = 1;
        admin = msg.sender;
    }
    function withdraw() external {
        require(msg.sender == admin);
        payable(admin).transfer(this.balance);
    }
}
```

Figure 1: Motivating Example

of Python code and 2,800 lines of JavaScript code, and used it to fix 12,526 real-world upgradeable contracts that are potentially susceptible to storage collision issues. Our evaluation has demonstrated that our patch code can precisely capture and disable actual storage collision attacks, while also avoiding unnecessary interference during normal contract operation. Our code, documents, and experimental data are publicly available at: https://github.com/scpatch/Storage-Collision-Patch

In summary, this work makes the following contribution:

- We study upgradeable contracts and gain crucial insights: to accurately identify storage collision issues, it is necessary to continuously monitor runtime storage accesses from both the proxy and logic contracts.
- We design and implement CollisionRepair, which inserts patch code into vulnerable contract bytecode to monitor runtime storage collisions. We developed a new mechanism for deploying and verifying patched contracts to mitigate collisions caused by storage accesses before and after patch deployment.
- We apply CollisionRepair to 12,526 real-world vulnerable contracts, effectively and efficiently neutralizing real attacks with reasonable overheads.

2 Problem Statement

2.1 Motivating Example

Vulnerable Upgradeable Contract. To motivate our research, we introduce an example of an upgradeable Ethereum smart contract, shown in Figure 1. This contract could lead to a "storage collision". Our example is adapted from one discussed in the prior work [42], but we have modified it to highlight an issue that has not been adequately explored in the existing literature. This example illustrates an upgradeable contract consisting of a user-facing `Proxy` contract and a backend `Logic` contract. Both contracts inadvertently assign variables of different types to the same storage locations, potentially leading to security issues.

More concretely, the `Proxy` contract allocates two three-element arrays at the beginning of its storage: `uint64[3] fees` (ln.2) and `address[3] feeRecipients` (ln.3). In Ethereum, each storage slot can hold 32 bytes; thus, the three unsigned integers in the `fees` array are stored together in slot 0, while each 20-byte address in the `feeRecipients` array occupies its own slot. These arrays are retained for legacy purposes and represent the percentages and recipients for three types of fees within the contract: penalty fee, transaction fee, and sales fee (ln.7). Although the `fees` array is deprecated, the developer accidentally left the `updateFee()` function (ln.14) active, allowing it to be called by users to change the array content and thus lead to errors. This upgradeable contract is initialized by the `constructor()` (ln.9), which initializes the `feeRecipients` (ln.10) and calls the initialization function `LOGIC.initialize()` (ln.12) from the logic contract.

The `Proxy` contract acts as the consistent external interface for this upgradeable contract. When it receives a user request for a function not defined within itself, it processes the request using its `fallback()` function (ln.17) and utilizes a `delegatecall()` (ln.18) to forward the call to the `Logic` contract (ln.27). The key distinction between a delegate call and a regular call is that the former executes within the caller's storage without changing the calling context, whereas the latter switches to the callee's context, operating within the storage of the callee's host contract. Consequently, since all functions defined in the backend `Logic` contract are invoked through the frontend `Proxy` using `delegatecall()`, they in effect share the same storage space with the `Proxy`.

Nevertheless, the `Logic` contract defines several variables stored in the same memory space. To prevent them from being allocated at the same storage locations, the developer inserted padding bytes—in the form of an array `uint256[256] __gap` (ln.28)—equating to 32×256 bytes before defining other variables in the `Logic` contract, such as `admin`, `artworkIDs`, and `artworkHolders` (ln.29-31). This padding ensures these variables do not collide with those defined in the `Proxy`. However, the developer overlooked that the `Logic` inherits from an abstract contract named `Initializable` (ln.22), which defines a Boolean variable `initialized` (ln.23). Due to this inheritance, `initialized` is allocated at the very top of the storage, preceding the `__gap`. As a result, there is an overlap in storage between `fees` from `Proxy` and `initialized` from `Logic`, with both starting at offset 0 in slot 0. The former occupies 8×3 bytes, while the latter Boolean variable consumes one byte.

Normally, modifying the `initialized` variable to 1 (ln.35) prevents the `initialize()` function from being executed again, as the condition `require(!initialized)` (ln.34) will not be met. However, if the `updateFee(FeeType.penalty, 0)` function (ln.14) is inadvertently called, it can reset the lower 8 bytes—including `initialized`—to zeroes, thereby re-enabling the `initialize()` function. If an attacker then triggers the `initialize()` function, he can successfully change the `admin` variable to their own address (i.e., `msg.sender`, ln.36), giving him the ability to withdraw all the funds from the contract (ln.40).

Limitations of Static Detectors. CRUSH [42] can detect the potential storage collision issue, as the `Proxy` and `Logic` contracts might access the same storage locations but interpret the data differently (`uint64` vs. `bool`). However, such a type mismatch does not necessarily lead to a storage collision if the actual accesses by the contracts do not coincide. **For instance, if the index `i` into the array `fees[i]` is derived from a runtime input that, due to input validation at the web interface, can never be 0, it will not affect the lowest byte. Under such conditions, the "potential" security risk would not be realized.**

The fundamental issue stems from the limited program modeling and over-approximated results produced by static analysis. As a result, when a "may-happen" security bug is reported to contract developers, they must meticulously verify the vulnerability's existence before developing a patch. Given the time-consuming nature of this process, there is a significant need for a quick first-aid solution that can mitigate potential risks promptly. Multiple studies, including [25, 33, 40, 41, 57], recognize this challenge and propose automated techniques to monitor smart contract runtime behaviors. These techniques aim to accurately block observed exploits while permitting legitimate execution. Specifically, more recent work, such as EVMPATCH [41], SMARTSHIELD [57], ELYSIUM [25], employs a bytecode rewriting-based approach to insert additional checks into original programs. These security checks can detect and disable atomicity violation (reentrancy bugs [35]), integer overflow [46], and missing access control [25].

Limitations of Existing Patching Techniques. However, the nature of storage collision vulnerabilities significantly differs from the security issues mentioned previously, making them resistant to correction using existing patching tools.

Stateless Detection. The security problems highlighted in prior studies are typically identifiable through stateless observation of a single execution point. For instance, a reentrancy attack can be detected and thwarted by verifying if a caller function has acquired the reentrant lock without needing to

```
if (gasleft() <= 2300) {
  return;
}

address target = target_;
bytes memory data = msg.data;
assembly {
  let result := delegatecall(gas, target, add(data, 0
       x20), mload(data), 0, 0)
  ...
}
```

Figure 2: Branch Condition based on Remaining Gas

know the call stack's state. Similarly, integer overflow issues can be mitigated by checking the integer's value before its use in critical operations like funds transfers, regardless of how the value was accumulated through prior arithmetic operations. Likewise, unauthorized function invocations can be controlled by verifying the identity of each caller, without inspecting their historical invocation pattern.

In contrast, uncovering a storage collision issue cannot be achieved by merely monitoring a single storage access event, such as the memory write `fees[t] = percent` itself. Storage collisions occur due to conflicts between current storage accesses and previous memory writes. Consequently, understanding the "ownership" status of the storage—indicating *who* wrote to *which* memory cell in *whose* storage and *when*—is crucial. Therefore, identifying a storage collision problem requires a state-aware approach, necessitating continuous monitoring of all storage accesses to track these state changes. Only then can we detect the storage collision in the motivating example, as the memory write `fees[FeeType.penalty] = percent` made by the `Proxy` contract occurs when the corresponding storage location has already been used by the `Logic` contract to store the variable `initialized`.

Simplistic Patch Deployment. The prior work falls short in not only generating but also deploying patch code to effectively address storage collision problems. Because storage collisions are stateful issues, any patched version must account for the current storage state, which is influenced by previous memory accesses. However, prior work often initiates a patched program from a clean state, failing to detect collisions that may arise during this transition.

Imprecise Patch Verification. Although prior work verifies the fidelity of patched contracts by checking for behavioral discrepancies before and after patching, its analysis is fundamentally imprecise due to the assumption that inserted code does not change the original contract states, such as account balances. However, this assumption is flawed—even if patch code is correctly implemented without explicitly accessing the original contract states, it can still implicitly affect them because the added gas cost may alter the program's control flow. As shown in Figure 2, in the real-world *bZx Vesting Token* contract [22], the program execution depends on the remaining gas—using `gasleft()`—which may cause it to take different branches, resulting in varied updates to global states such as the `data` variable.

Essentially, contract execution results in either (1) an EVM error due to insufficient gas or (2) a successful run. If the contract behavior depends on `gasleft()`, case (2) further splits into (2.1) the `gasleft()` condition not met and (2.2) the condition met. While increasing gas reveals both (1) and (2), distinguishing (2.1) from (2.2) is more difficult due to narrow path conditions. We confirmed this across all 1,185 contracts containing `gasleft()` checks. For example, in Figure 2, the gas range for case (2.2)—(0, 2300]—is much narrower than that for (2.1)—(2300, block.gaslimit)—making it less likely to be triggered.

2.2 Problem Statement

To overcome the limitations of previous efforts, we introduce CollisionRepair—an automated approach for mitigating storage collision issues in smart contract bytecode. We argue that this technique must meet three design requirements.

(1) **Bytecode Rewriting.** Our patching technique must operate directly on the compiled bytecode of smart contracts. The patch code is statically inserted into the original bytecode programs. We avoid using modified compilers to add patch code to source code, which may not always be available, and we do not use emulators for runtime instrumentation, as this introduces significant runtime overhead.

(2) **Accurate Intervention.** Our inserted patch code must collect the runtime information necessary to determine the presence of storage collisions. It should precisely block operations that are about to cause these collisions, while permitting the legitimate execution of contract programs.

(3) **Practical Solution.** The patching technique should automatically generate and deploy patched code, requiring minimal developer involvement. Executing the patch code should not alter contract behavior or lead to significant overheads.

Given the design requirements, we develop our patching mechanisms based on the following threat model.

Threat Model. Our trusted computing base includes *(a)* blockchains, *(b)* smart contract runtime, *(c)* communications between smart contracts, and *(d)* transaction history.

We assume that blockchain algorithms and infrastructure are secure. Attacks aimed at exploiting vulnerabilities in the consensus mechanism [27], mining [24], or peer-to-peer networks [29] are outside the scope of this paper. We do not consider lower-level software attacks that compromise the integrity of operating systems and smart contract runtimes; nor do we consider network-level attacks that disrupt or intercept network traffic. We also do not cover price manipulation [52] and front-running attacks [58] that exploit economic system dynamics rather than software vulnerabilities. Additionally, we assume that past transactions are securely documented and maintained.

However, we do address application-level issues arising from the implementation of smart contract code. Smart contracts may be poorly designed or insecurely implemented by

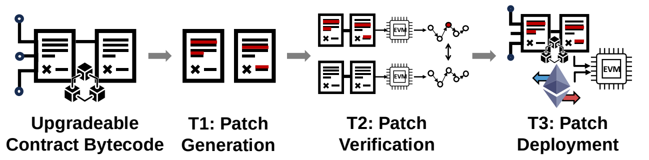


Figure 3: Architectural Overview

developers, potentially leading to unintended security vulnerabilities. These vulnerabilities can be exploited by malicious users to abuse benign contracts, causing financial losses to other users or contract owners.

2.3 Approach Overview

To achieve the aforementioned design goals, we propose the design of CollisionRepair. This system takes as input the bytecode programs of upgradeable contracts from the Ethereum blockchain, and outputs patched contracts that run on the blockchain while preserving the existing contract state. To accomplish this, we have outlined three key tasks, as illustrated in Figure 3.

- **T1: Patch Generation.** Given the proxy and logic contracts of an upgradeable contract, we first incorporate patch code into the original contracts through *static bytecode instrumentation*. This instrumented code continuously monitors and verifies every storage access made by both contracts. It also keeps track of the call stack to determine the initiator of each storage access and the targeted storage space—the "*ownership*" of individual storage locations—and controls memory accesses that will lead to collisions by reverting the current transaction.
- **T2: Patch Verification.** After creating a patch, it is essential to verify that the patched version does not disrupt original legitimate operations. To do this, we use differential testing to compare execution traces before and after patching. Specifically, we model semantic equivalence of execution traces based on state-changing events, such as function calls and storage writes, while accounting for the patch code's implicit impact on contract behavior due to altered gas consumption.
- **T3: Patch Deployment.** Once a patched upgradeable contract passes verification, we deploy both the patched proxy and logic contracts to the blockchain. To restore the current "ownership" state of storage, we replay past transactions on the instrumented code in emulated environments. This allows the patched code to operate with the recovered "ownership" state rather than starting from a blank state.

3 Patch Generation

3.1 Design

Formal Model. To effectively monitor and disable the storage collision problem at runtime, we first formally define this issue as an "ownership" verification problem.

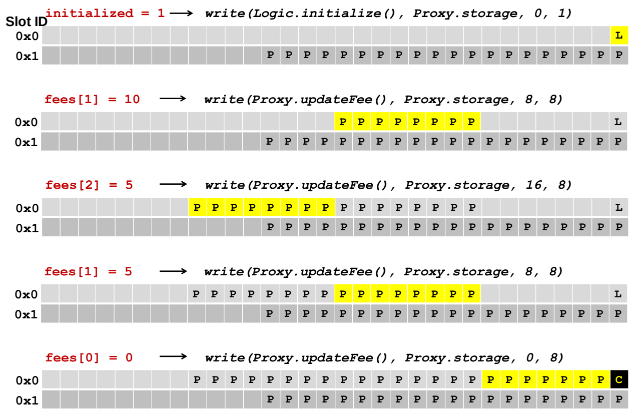

Slot ID
initialized = 1 ⟶ write(Logic.initialize(), Proxy.storage, 0, 1)
0x0
0x1
fees[1] = 10 ⟶ write(Proxy.updateFee(), Proxy.storage, 8, 8)
fees[2] = 5 ⟶ write(Proxy.updateFee(), Proxy.storage, 16, 8)
fees[1] = 5 ⟶ write(Proxy.updateFee(), Proxy.storage, 8, 8)
fees[0] = 0 ⟶ write(Proxy.updateFee(), Proxy.storage, 0, 8)
P
L
C

Figure 4: Ownership Changes for the Motivating Example

Definition 1: The **ownership** of a storage cell is defined as the entity that wrote to this cell. The **ownership** of a storage space encompasses the ownership status of all memory cells within that space. Ownership is exclusive, meaning each memory cell is owned by only one entity.

At a high level, we adapt the concept of "ownership" from Rust [43], which utilizes this principle for safe memory management. We apply this concept in a simpler and more restrictive way. Each memory location can be written to by only one entity, and the writer permanently owns that memory space—we do not support "drop" or "borrow" functionalities. This approach ensures a strict check for storage collisions: once a memory location is occupied by an owner, any access (write or read) to the same location by other entities is considered a collision, irrespective of whether the owner still actively uses the memory content.

Note that intentional storage reuse may lead to logic-level false positives under our model. However, we argue that reusing the same slot for both security-sensitive and non-sensitive data is inherently risky and warrants an alert. In practice, we observed no such false positives.

Figure 4 illustrates how the ownership status of the storage space changes over time in the motivational example. Specifically, it shows the first two slots (`0x0` and `0x1`) of the storage space in the proxy contract. Initially, the `feeRecipients` array (comprising three 20-byte addresses) is updated by the `Proxy` contract, making the proxy (**P**) the owner of the lower 20 bytes of slots 1 to 3. Subsequently, the `initialized` flag is modified by the `Logic` contract, resulting in the ownership of the lowest byte in slot 0 being transferred to the `Logic` (**L**). During operation, users may inadvertently activate the `updateFee()` function via the `Proxy` contract. Depending on the arguments supplied, specific memory cells will be owned by the proxy. For example, when `fees[1] = 10` and `fees[2] = 5` are executed, the proxy assumes ownership of bytes 8 – 15 and bytes 16 – 23, respectively. Although the `Proxy`'s `fees` array overlaps with the `Logic`'s `initialized` flag, these actions do not breach our ownership rules. Moreover, the proxy overwriting bytes 8 – 15, an area it owns, does not cause a collision. However, when the proxy writes to `fees[0]`, it disrupts the ownership of the lowest byte in slot 0, thereby indicating a storage collision issue.

Note that we do not monitor the specific content of storage. Our focus is on ownership violations, which prompt us to issue a collision warning. While it is true that security issues may arise when a memory location is set to a specific value, the potential for risk extends beyond that. For example, in the motivational example, a security problem occurs only if the `initialized` flag is set to 1, which would allow attackers to reset the contract's admin. However, we argue that even if this variable is set to a different value, the conflict arising from overlapping memory writes by different contracts represents a fundamental risk. Therefore, such conflicts need to be preemptively addressed and prevented.

Patching Algorithm. From Figure 4, it is evident that to accurately monitor changes in ownership status, we need to examine every storage access and record five essential parameters: ① *access type* (read or write), ② *host contract*, ③ *target storage*, ④ *target memory address*, and ⑤ *data size*. For example, when the `initialized` variable is modified, the operation is a `write`, triggered by the `initialize()` function of the `Logic` contract. This action targets the storage space of the `Proxy` contract because the `Logic.initialize()` function is executed within the context of the `Proxy`, due to a delegate call from the latter to the former. The operation writes to the `initialized` variable at offset 0 in slot 0, affecting 1 byte. Thus, our patch code must capture such details—for instance, ***write****(****Logic****.*`initialize`*,* ***Proxy****.*`storage`*,* ***0****,* ***1****)*—for each memory access.

To achieve this, we have developed a patching algorithm illustrated in Algorithm 1, designed to insert patch code into the bytecode of vulnerable smart contracts. This code is responsible for collecting aforementioned information so as to block memory accesses that result in storage collisions.

Our algorithm, InsertPatchCode(), takes an original smart contract, `SC`, either a proxy or a logic contract, as input and inserts patch code into its bytecode instructions. For a given `SC`, we first insert code in its constructor to establish the initial ownership states (`owner`) for the storage spaces of both the proxy and logic contracts (ln.2). Additionally, we incorporate code to initialize an empty map $\mathbb{CS}$ of call stacks (ln.3). Each entry maps an individual contract execution—indicated by its transaction (or message) `sender`—to its runtime call stack. An item in the stack comprises a pair $\langle$`contract`, `storage`$\rangle$, and the stack is used to track these contexts (i.e., ② and ③) throughout the contract execution.

To initialize an individual call stack, we insert instructions at the entry point of every function `func` (ln.4), since any function could be directly triggered by external transactions. To do so, we first obtain the current message `sender` (ln.5), then verify its existence in $\mathbb{CS}$ and create a new call stack `cs` if none exists or if an existing one is empty (ln.6). Finally, we push the new context $\langle$`func.contract`, `func.storage`$\rangle$ into `cs` (ln.7).

Algorithm 1 Patch Code Insertion

```
 1: procedure InsertPatchCode(SC)
 2:     owner $\leftarrow$ AddInitOwnership(SC)
 3:     $\mathbb{CS}$ $\leftarrow$ AddInitCallstackMap(SC)
 4:     for $\forall$func $\in$ SC do
 5:         sender $\leftarrow$ AddGetMsgSender()
 6:         cs $\leftarrow$ AddNewCSIfNotExistOrEmpty($\mathbb{CS}$, sender)
 7:         AddPush(cs, $\langle$func.contract, func.storage$\rangle$)
 8:     end for
 9:     for $\forall$instr $\in$ SC do
10:         if instr is "CALL" or "DELEGATECALL" or "RETURN"
11:          or "SLOAD" or "SSTORE" then
12:             sender $\leftarrow$ AddGetMsgSender()
13:             cs $\leftarrow$ AddGetCS($\mathbb{CS}$, sender)
14:         end if
15:         if instr is "CALL" then
16:             tgt $\leftarrow$ AddGetTarget(instr)
17:             AddPush(cs, $\langle$tgt, tgt.storage$\rangle$)
18:         else if instr is "DELEGATECALL" then
19:             tgt $\leftarrow$ AddGetTarget(instr)
20:             ctx $\leftarrow$ AddPeek(cs)
21:             AddPush(cs, $\langle$tgt, ctx.storage$\rangle$)
22:         else if instr is "RETURN" then
23:             AddPop(cs)
24:         else if instr is "SLOAD" or "SSTORE" then
25:             ctx $\leftarrow$ AddPeek(cs)
26:             op $\leftarrow$ AddGetOperands(instr)
27:             AddCheckOrEditOwner(owner, ctx, op)
28:         end if
29:     end for
30: end procedure
```

Our patching algorithm then examines every single bytecode instruction, `instr`, in the contract (ln.9), and inserts additional instructions before those identified as relevant. Relevant bytecode instructions include `CALL`, `DELEGATECALL`, `RETURN`, `SLOAD` and `SSTORE`. If `instr` falls into this category (ln.10-11), we add instructions to retrieve the corresponding call stack `cs` from the $\mathbb{CS}$ (ln.13) according to the current message sender `sender` (ln.12).

Additionally, when `instr` is a `CALL` (ln.15) or a `DELEGATECALL` (ln.18), we add instructions to push the pair $\langle$`contract`, `storage`$\rangle$, representing the call target address and the storage it utilizes, onto the call stack `cs` (ln.17, ln.21). This requires inserting instructions to extract the call target `tgt` in both scenarios (ln.16, ln.19). In contrast, in these scenarios, different storage contexts are used: A `CALL` instruction triggers a context switch to the storage of the call target, `tgt.storage`, whereas a `DELEGATECALL` retains the current storage context `ctx.storage`, as indicated by the top entry of the call stack (ln.20). Upon encountering a `RETURN` instruction (ln.22), we insert a "pop" instruction to restore the previous context by removing the current one from `cs` (ln.23).

We also integrate ownership verification code prior to the `SLOAD` and `SSTORE` instructions (ln.24). This involves inserting code to first check the top of the call stack (ln.25) to ascertain the current context `ctx`. Subsequently, we retrieve the operands (`op`) related to the memory accesses (ln.26)—④

address and ⑤ *size*. This information is then passed to a check/edit function (ln.27). Specifically, for each read, we insert instructions to verify the absence of any collisions; for each write, we both check for collisions and, if no issues are found, update the `owner` status. If a collision is detected, the check/edit functions trigger a `revert()` function to roll back the current transaction.

Motivating Example. We demonstrate the patched code for our motivating example in Appendix A.

3.2 Implementation

Although the example patch code appears straightforward, implementing it at the bytecode level is complex and requires careful consideration to address several challenges.

(1) Trampoline-based Instrumentation. The first hurdle is fixing jump targets. In EVM bytecode, all instructions from different functions are organized in a single flat memory space, and jump targets are relative offsets. Inserting patch code affects these jump targets, which must be corrected afterwards. However, the frequent use of computed jumps makes adjusting these offsets challenging, as it requires identifying and modifying the calculation parameters in the stack.

To address this, we use a trampoline-based method. This technique appends the patch code as a new basic block at the end of the original code section, preserving the original offsets. All code in the old basic block that includes instructions requiring monitoring is relocated to the new appended block, where monitoring and enforcement code is inserted before key instructions. A jump is placed at the end of the new block to transfer control to the original block's successor, while a jump to the patch code is inserted at the start of the old block. The remaining space in the old block is filled with "`INVALID`" instructions to maintain its original size. EVMPatch [41] employs a similar approach, but our work is distinguished by the specific information identified for addressing storage collisions and the methods developed to integrate it into the "trampoline", as detailed below.

(2) Integrating Ownership States to Patched Contracts. Implementing ownership states and the functions to check and update these states is non-trivial within our design. Although it is theoretically possible to embed these data structures and functions directly in the patched contracts, adding such extensive bytecode is complex. As a practical alternative, we implement the ownership map and state check/update functions in Solidity source code within an external monitoring contract, with the patch code in the main contract making cross-contract calls to manage and verify ownership states. While this approach simplifies implementation, it may incur additional latency and gas costs. We evaluate runtime and gas overheads to understand system practicality in Section 6.

(3) Identifying Call Targets. To track the call stack, we need to capture the addresses of target contracts by extracting arguments from the `CALL` and `DELEGATECALL` instructions,

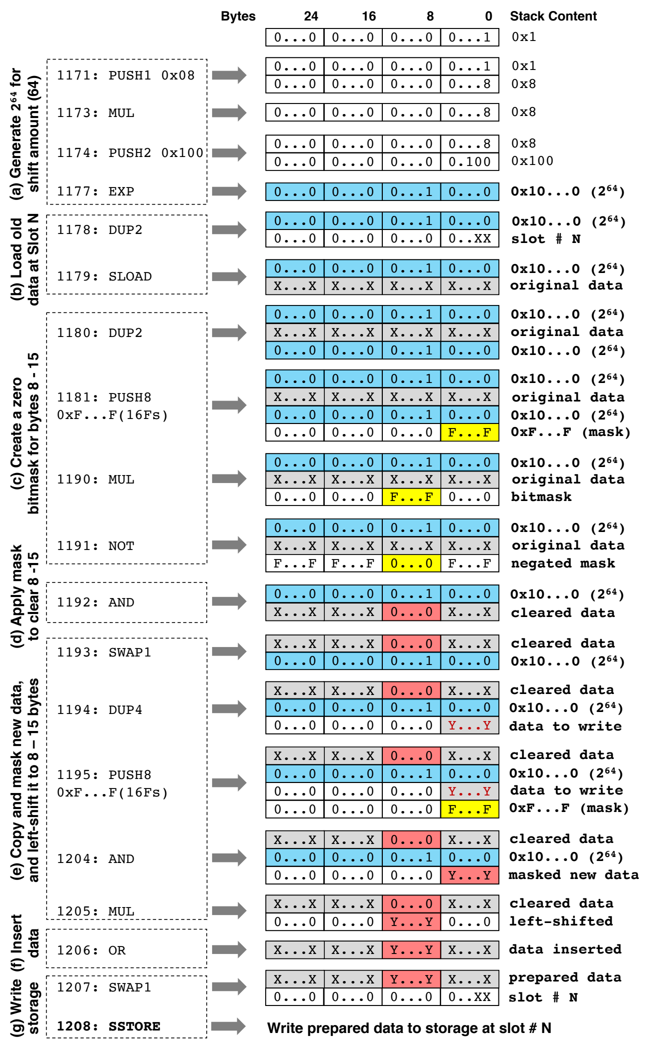


Figure 5: Preparation of `SSTORE` Operands

and thus must understand their calling conventions. Specifically, both instructions pull inputs from the EVM's *memory* and *stack*. The *memory* holds the target function's signature (called selector) and function arguments, while the *stack* contains environmental variables, such as the *gas* amount, *calldata*, and *return data* associated with the transaction. In both cases, the *address* of the target contract is the second environmental variable on the stack. We therefore insert code to monitor the stack state before each call to capture it.

(4) Extracting Operand Values for Memory Access Bytecode. In source code, the ④ *target memory address* and ⑤ *data size* are explicitly specified for memory reads and writes. In bytecode, however, this information is implicit, requiring an understanding of Ethereum Virtual Machine (EVM) conventions for memory access instructions such as `SSTORE` and `SLOAD`. We examined Solidity versions 0.4.x – 0.8.x with three optimization settings—200 (default), 800 (used by Uniswap)—and 2000 (aggressive), and found all consistently use the same *mask-and-shift* pattern for `SSTORE` parameters.

Specifically, the EVM always reads or writes an entire 32-byte slot. If only part of a slot is modified, it first loads the full slot, applies bit masks to update the relevant bytes, and then writes the modified 32-byte value back. Thus, accurately determining the position and size of modified bytes requires insight into EVM's bit masking operations.

Figure 5 provides an example of bytecode instructions for writing to storage. On the left is the bytecode sequence that prepares data for the `SSTORE` instruction, located at the end. For each bytecode instruction, the affected stack state is shown on the right. Each slot in the stack is 32 bytes long, and the stack grows downward, with only the top few slots displayed.

Particularly, this instruction sequence writes to bytes 8 – 15 of the 32-byte Slot # N in storage. This process takes approximately 7 steps. **(a)** The first four instructions, from `PUSH1` at offset 1171 to `EXP` at 1177, generate the number 2^{64}, indicating a 64-bit (8-byte) shift from the least significant bit to reach byte 8. **(b)** The next instruction, `DUP2` (1178), duplicates the target slot number (N) already on the stack and moves it to the top, enabling the upcoming `SLOAD` instruction (1179) to load the original data from Slot # N onto the stack. **(c)** Next, instructions 1180 to 1191 generate an 8-byte zero mask at bytes 8 – 15: they first create an 8-byte ones mask at bytes 0 – 7 using `PUSH8 0xF...F` (1181), then left-shift this mask to bytes 8 – 15 by multiplying it by 2^{64} (1190), and finally negate the entire slot to produce the desired zero mask. **(d)** Then, an `AND` instruction (1192) applies the zero mask to the original data, clearing bytes 8–15. **(e)** With the original data masked, a series of instructions (1193 – 1205) copies the new data to the top of the stack (1194), applies a ones mask (1195 and 1204), and shifts the data into bytes 8 – 15 (1205). **(f)** The new data is then inserted into the original data at bytes 8 – 15 using an `OR` operation (1206). **(g)** Finally, an `SSTORE` instruction writes the modified data back to the same slot, completing the storage update (1207 and 1208).

From this typical sequence, we observe that the target memory address (to be more precise, the offset within a specified slot) corresponds to the shift amount (64), while the data size is related to the bitmasks used to clear or preserve specific bits. To efficiently obtain this information, we can inspect the stack at the point when both pieces of data are present. In practice, we monitor the stack state immediately before the `1204:AND` instruction, the last point where both pieces of data are available—at the top and third stack slots, retrievable using `DUP1` and `DUP3`, respectively.

As discussed earlier, we use the trampoline-based method, copying all instructions in this basic block into a new basic block. In the new block, located at the end of the code section, we insert the patch code partially shown in Figure 6. This code snippet begins with the relocated `PUSH8` instruction (originally at offset 1195, now at 1215), copies the slot number, bitmask and shift amount from the stack using `DUP1` (1256), `DUP1` (1224) and `DUP3` (1232), stores them in *memory* as the first, second and third arguments of the "`check()`" function

```
...                     // Relocated instructions from top
1215: PUSH8 0xFFFFFFFFFFFFFFFF  // Relocated instruction
1224: DUP1              // Duplicate the bitmask (0xF...F)
...                     // Locate the memory for 2nd arg
1231: MSTORE            // Store the bitmask as 2nd arg
1232: DUP3              // Duplicate the 2^64
...                     // Locate the memory for 3rd arg
1239: MSTORE            // Store 2^64 as 3rd arg
1240: PUSH4 0x56B10083// Selector of ``check()'' function
...                     // Store the function selector
1252: AND               // Relocated instruction
...                     // Relocated instructions
1255: SWAP1             // Relocated instruction
1256: DUP1              // Duplicate the slot number
...                     // Locate the memory for 1st arg
1263: MSTORE            // Store slot number as 1st arg
...                     // Prepare for other arguments
1275: PUSH20 0xFE..1EE// The monitoring contract address
1296: GAS               // The remaining gas
1297: CALL              // Call the ``check()'' function
1298: SSTORE            // Relocated instruction
1299: POP               // Pop ``check()'' return value
...                     // Relocated instructions
1303: JUMP              // Jump to next basic block
```

Figure 6: Partial Patch Bytecode to Monitor `SSTORE`

via `MSTORE` (1263, 1231 and 1239), prepares additional arguments such as saving `0x56B10083` (the function selector of "`check()`") (1240) and pushing the address of the monitoring contract onto the stack (1275), and finally calls the "`check()`" function (1297). More complete code is in Appendix B.

4 Patch Verification

Basic Idea. Once the patch code is generated, we must verify that it does not disrupt the original functionalities before applying it to the corresponding real-world contract. To do so, we use differential testing to determine semantic equivalence between the original and patched contracts during benign transactions. To achieve this, we can deploy both the unpatched and patched contracts in a testing environment and replays past transactions on each to observe equivalence in their EVM bytecode execution traces.

However, the two execution traces are unlikely to be identical due to subtle runtime differences. To address this, prior work, EVMPatch [41], proposed modeling the semantics of execution traces by focusing on key state-changing instructions, such as `SSTORE` and `CALL`, which significantly impact storage states and calling contexts, respectively.

Formally, a *state* s for a bytecode instruction sequence I is a 3-tuple (val, pc, mem), where $val : V \rightarrow Values$ is an assignment of values to the variables V; pc is a value for the program counter; and $mem : Addr \rightarrow Values$ gives the memory contents. $mem[a]$ denotes the value stored at address a. If we execute an instruction i from I in state s_0, we transition to a new state s_1 generating an event e. An event can be function call `CALL` or a storage write `SSTORE`. If an instruction i does not generate such an event, $e(i) = null$. Then, we denote a state change from s_0 to s_1 that creates an event e as $s_0 \xrightarrow{e} s_1$.

Hence, the prior work in fact defines semantic equivalence between pre- and post-patched contracts as follows:

Definition 2: The original and patched contracts exhibit the same normal behavior if their EVM bytecode execution traces, I_O and I_P, are finite instruction sequences, and there exist two contract program states, s_0^O and s_0^P, such that $mem(s_0^O) = mem(s_0^P)$ (i.e., the memory in both states is identical), and the following two conditions hold:

- **Condition 1:** Let the two state transition sequences be given as follows:

$$\sigma(I_O, s_0^O) = s_0^O \xrightarrow{e_1^O} s_1^O \xrightarrow{e_2^O} \dots \xrightarrow{e_j^O} s_j^O$$

$$\sigma(I_P, s_0^P) = s_0^P \xrightarrow{e_1^P} s_1^P \xrightarrow{e_2^P} \dots \xrightarrow{e_k^P} s_k^P$$

  Let $affected(\sigma(I_O, s_0^O))$ be the set of addresses a such that $mem(s_0^O)[a] \neq mem(s_j^O)[a]$. Thus, $affected(\sigma(I_O, s_0^O))$ is the set of memory addresses whose value changes after running the bytecode instructions from the initial state. These memory regions include the call stack and persistent storage, which are affected by `CALL` and `SSTORE` from the original unpatched instructions, respectively. We require that $mem(s_j^O)[a] = mem(s_k^P)[a]$ holds for all $a \in affected(\sigma(I_O, s_0^O))$. i.e., values at addresses belonging to $affected(\sigma(I_O, s_0^O))$ are the same after executing the two bytecode sequences.

- **Condition 2:** If $pc(s_j^O) \in affected(\sigma(I_O, s_0^O))$, then

$$pc(s_j^O) \in affected(\sigma(I_O, s_0^O)),$$

  i.e., if the program counter at the end of executing I_O points to the affected memory area, then the program counter after executing I_P should also point to the affected memory.

While **Condition 1** ensures dataflow consistency, **Condition 2** ensures that the control flow remains intact. To verify identical impacts on data and control flow, prior work compares the equivalence between two event sequences, $e_1^O \to e_2^O \dots \to e_j^O$ and $e_1^P \to e_2^P \dots \to e_k^P$. Specifically, they check that the non-null events (i.e., `SSTORE` and `CALL`) in both sequences follow the same order and that each pair of corresponding function events has the same input and output.

Our Enhanced Approach. However, this definition assumes that the patch code does not alter the original program states. While this assumption appears reasonable—since correct patch code should not interfere with the original program logic, such as by calling original contract functions or accessing its storage—it may not hold if the patch code implicitly affects program states by increasing gas consumption, as depicted in Figure 2. In such cases, even though the patch code does not interfere with the original contract logic, the execution trace of the patched contract does not align with the original, according to the patch testing used in the prior work.

We argue that the patch code can still be considered correct despite changes to program states. This is because proactively checking the remaining gas with `gasleft()` is often used as an intentional mechanism to gracefully revert transactions, representing legitimate operations within normal contract logic rather than disrupted functionality.

Algorithm 2 Patch Code Verification

```
 1: procedure VerifyPatch(O, P, $\mathbb{TX}$)
 2:     for $\forall$tx $\in \mathbb{TX}$ do
 3:         $\sigma(I_O, s_{tx_0}) \leftarrow$ Replay(O, tx)
 4:         $\sigma(I_P, s_{tx_0}) \leftarrow$ Replay(P, tx)
 5:         while IsNotEmpty($\sigma(I_O, s_{tx_0})$) do
 6:             $e^O \leftarrow$ Dequeue($\sigma(I_O, s_{tx_0})$)
 7:             if ($e^O$ is "CALL" $\vee$ $e^O$ is "SSTORE") $\wedge$
 8:               ("gasleft()" $\in$ GetDominator($e^O$, O) $\wedge$
 9:               $e^O \notin$ GetPostDominator("gasleft()", O)) then
10:                 $e^P \leftarrow$ DequeueUntilFound($\sigma(I_P, s_{tx_0}), e^O$)
11:                 if $e^P$ = null then
12:                     return false
13:                 end if
14:                 if GetInOutStates($e^O, \sigma(I_O, s_{tx_0})$) $\neq$
15:                   GetInOutStates($e^P, \sigma(I_P, s_{tx_0})$) then
16:                     return false
17:                 end if
18:             end if
19:         end while
20:     end for
21:     return true
22: end procedure
```

To more precisely verify the patched contracts, we propose modifying Definition 2 by replacing $affected(\sigma(I_O, s_0^O))$ with $core(\sigma(I_O, s_0^O))$. Here, $core(\sigma(I_O, s_0^O))$ represents a core subset of memory addresses that change due to key events, excluding memory regions affected by instructions that are conditionally executed based on `gasleft()` checks.

Consequently, our patch verification process can be illustrated by Algorithm 2. Our algorithm takes three inputs: the original and patched contracts, `O` and `P`, along with the past transactions $\mathbb{TX}$ of the original contract. It outputs a Boolean value to indicate whether the two contracts exhibit identical behavior for any transaction `tx` in $\mathbb{TX}$ (ln.2).

For each `tx`, we first replay it using the Ganache [13] emulator on both contracts, starting from the same initial state s_{tx_0}, and generate their execution sequences $\sigma(I_O, s_{tx_0})$ and $\sigma(I_P, s_{tx_0})$ (ln.3-4). We then dequeue each event e^O from the original execution sequence $\sigma(I_O, s_{tx_0})$ until the sequence is empty (ln.5-6). If an event e^O is either `CALL` or `SSTORE` (ln.7) and is within a branch guarded by a condition check with `gasleft()`—specifically, `gasleft()` is a dominator of e^O (ln.8) and e^O is not a post-dominator of `gasleft()` (ln.9)—we locate the corresponding event e^P at the beginning of the new sequence $\sigma(I_P, s_{tx_0})$ and dequeue any preceding events (ln.10).

If a non-null e^P is not found, we return false, indicating verification failure (ln.11-13). Otherwise, we extract the input and output states of both events (ln.14-15). If the input or output states do not match, we again return false (ln.16). Here, we specifically inspect storage states for `SSTORE` events and call stack states for `CALL` events. If every transaction is successfully verified on the contracts, we return true (ln.21), indicating the patched code preserves the original behavior.

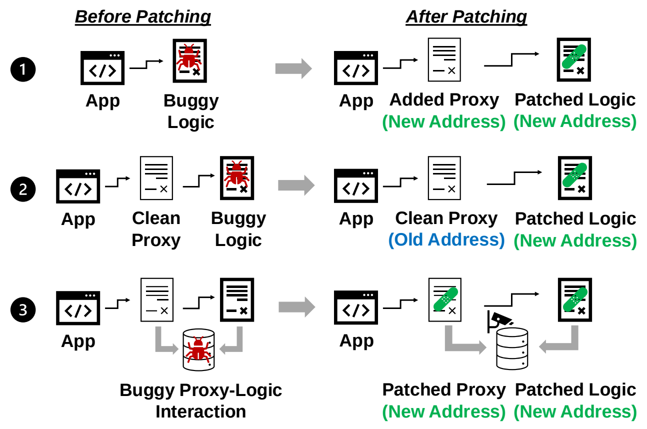


Figure 7: Deployment Scenarios

5 Patch Deployment

Once a patched contract is verified, we deploy it to the blockchain. Additionally, unlike prior work that executes the patched contract from a fresh state, our patch code must start with the current "ownership" state to account for any storage collisions between pre- and post-patching storage accesses.

Deployment Scenario. In fact, our deployment scenario differs significantly from that in prior work. Figure 7 illustrates the three deployment scenarios for patch code.

The first scenario, used in EVMPATCH, addresses vulnerabilities in a single, non-upgradeable contract. To replace a vulnerable contract, EVMPATCH converts the original simple contract into an upgradeable one by introducing a new proxy at a new blockchain address, with the patch program referenced through this proxy.

The second scenario illustrates the typical updating process in upgradeable contracts. It addresses vulnerabilities in upgradeable contracts, but only when the vulnerability exists within a single logic contract. Here, since the proxy contract is free of vulnerabilities, it can be reused. The proxy remains at its original address, while its reference to the vulnerable logic contract is updated to point to the new patched version.

The third scenario, which applies to our case, involves vulnerabilities in the interaction between the proxy and logic contracts due to storage collision issues. Here, the patch code must be applied to both contracts. Consequently, we replace both the original proxy and logic contracts with patched versions deployed at new addresses.

Ownership State Construction. To establish the current ownership state, we replay all past transactions on the patched version within the Ganache emulator, enabling our patch code to keep track of ownership transitions. If our replay detects any storage collision issues that occurred in the past, we will report the issue and halt deployment. Otherwise, we proceed to deploy the patched code to the blockchain, maintaining the recovered storage states.

Note that historical transactions may interact with external environments, such as reading or modifying blockchain or external account states. To accurately reproduce the impact of these transactions, it is essential to replay them within the precise historical blockchain environment. To achieve this, we use the Infura [14] testing framework, which can fork a snapshot of the entire Ethereum blockchain at a specified time. For each historical transaction, we first look it up on Etherscan [2] to determine the block number when the transaction occurred. Then, we use Infura to replicate the Ethereum environment for that specific block number and deploy the patched contracts into this environment. This allows us to run the transaction over our patched version in the snapshot while enabling the patched code to interact with the blockchain context.

6 Evaluation

We evaluate COLLISIONREPAIR across several dimensions. **(1) Correctness:** We verify that our patched contracts can be successfully deployed and executed on the blockchain, without introducing errors. **(2) Effectiveness:** We assess COLLISIONREPAIR's ability to block attacks while preserving normal functionality in the absence of attacks. **(3) Overhead:** We measure the runtime and gas overheads introduced by our patch code. **(4) Case Study:** We analyze specific cases to gain deeper insights into COLLISIONREPAIR's applicability.

6.1 Experimental Setup

Dataset. To conduct our evaluation, we first collected real-world smart contracts, divided into two categories:

(a) Random Samples: This dataset consists of 6,018 upgradeable contracts randomly selected from Etherscan [2]. To randomly select samples, we use a GitHub repository (also used by USCHUNT) containing over 482,000 mainnet contracts ordered by address. We generate 34,000 random indices in the range [0, 482K] to sample contracts. For evaluation, contracts must (1) include source code for manual verification, and (2) allow successful transaction replay on our local mainnet fork using the current blockchain state. Each replay takes 0.65 seconds on average. Replay failures may result from dependencies on (1) invalid external contracts (e.g., due to selfdestruct), (2) past timestamps, (3) unavailable oracles, (4) whitelisted callers, or (5) insufficient balances. Contracts with incompatible dependencies are excluded, even if some transactions are replayable. After filtering, 6,018 contracts remain.

Based on their source code, these contracts have an average of 543 lines of code (LOC), with 20.5% exceeding 1,000 LOC. Each contract includes an average of 26.8 functions, and 17.6% have more than 50 functions. Additionally, the compiled bytecode programs contain an average of **4,816 instructions**. These samples are used to assess whether our patching technique can be applied successfully to a large number of smart contracts. Also, running these random contracts instrumented by our tool allows us to gauge the general runtime and gas overheads introduced by the patch code.

(b) Vulnerable Contracts: This dataset comprises upgradeable contracts identified with storage collision issues. We obtained 959 such samples from the authors of CRUSH [42]; 913 were confirmed to be vulnerable upon manual inspection, 32 were false positives, and 14 remain undetermined. After de-duplication, we categorized the 913 confirmed cases into 205 groups, each containing contracts with identical proxy code deployed at different addresses.

The 32 false positive cases were correctly implemented according to EIP-1967 [19] or the EIP-2535 Diamond standard [11], both of which address storage collisions. CRUSH missed these safe contracts due to its inability to analyze inline assembly (needed for EIP-1967) or complex mappings (used in Diamond contracts). Note that while these protocols help prevent storage collisions, correct implementation is essential to avoid vulnerabilities.

This dataset serves as the ground truth and is primarily used to assess the effectiveness of COLLISIONREPAIR. We also use it as a set of "microbenchmarks" to measure overhead when the patch code is actively triggered or when a real attack is present. Specifically, each proxy-logic contract pair is treated as a single test target, as it represents a version of an upgradeable contract. Since each of the 913 proxy contracts may be associated with multiple logic contracts over time, we identified a total of **12,526** such pairs.

These 12,526 contracts average 422 LOC and 16 functions, with the top 100 averaging 2,597 LOC and 82 functions. Three of them have been confirmed as exploited. These vulnerable contracts hold an average TVL of $68K, with the top 10 averaging $1.7M—the Audius hack alone caused a $6M loss. In comparison, deploying a patched contract costs just $58.50.

Testing Environment. We have implemented a prototype system in 3,367 lines of Python code and 2,800 lines of JavaScript code. We build our tools based upon the **Octopus** [1] framework. Our experiments are conducted on a server equipped with Intel Xeon Gold 6330 CPU @ 2.00GHz, 256GB memory, and Ubuntu 20.04 LTS (64bit).

6.2 Correctness

Increased Size after Patching. We use COLLISIONREPAIR to instrument all 6,018 randomly selected contracts to monitor their storage accesses, regardless of whether they contain storage collision vulnerabilities. Each contract was successfully instrumented, with the instrumented programs averaging 5,047 bytecode instructions—an average size increase of 4.8% compared to the original code.

Patch and Deploy Time. We successfully deployed all patched versions to an Infura fork of the Ethereum mainnet. Figure 8 shows the distribution of patching and deployment times. Note that deployment time does not include time spent verifying the patch, such as creating snapshots or replaying

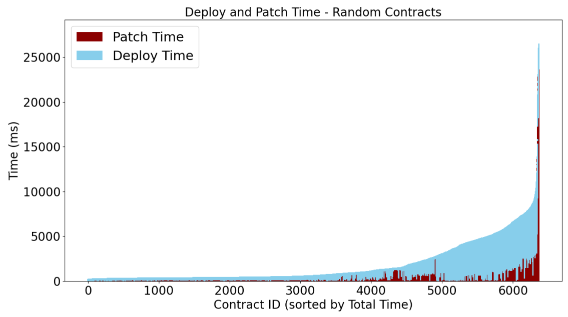


Figure 8: Patch and Deploy Time for Random Samples

transactions. The x-axis represents contract IDs, while the y-axis shows time in milliseconds. For each contract, patching time is shown in red and deployment time in blue. Contracts are sorted by total time in ascending order.

As shown, patching is very fast, averaging 874 milliseconds with a maximum of 23 seconds. Deployment time is slightly longer, averaging approximately 1 second, with the longest deployment taking only 8.8 seconds.

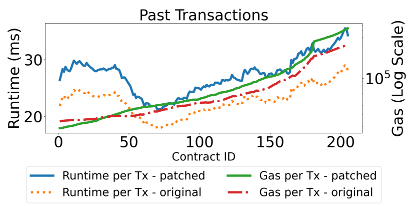
Transaction Replay. To ensure the patched code is error-free, we replay past transactions over the deployed contracts. Ideally, we would create an Ethereum fork for each transaction and replay it in the original environment, but this is tedious due to the high volume of transactions. Instead, we streamline the process by creating a single fork of the current Ethereum environment and re-running past transactions over the patched code in this setting.

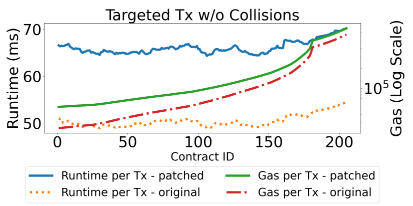
Replaying transactions in a modified environment may cause errors due to changes in blockchain and account states, rather than issues with the patch implementation itself. To address this, we first test on the original contract by deploying it in the same environment and re-running historical transactions. If a transaction succeeds, we then run it on the patched code. We re-run the first 1,000 transactions or all past transactions that successfully operate on the original contract over the patched version. In total, we replayed 591,786 transactions, averaging 98.3 transactions per contract. For 668 contracts (11.1%), we replayed over 200 transactions, and for 439 contracts (7.2%), we replayed over 500 transactions.

Our results show that all these transactions execute successfully on the patched contracts without errors. However, because we lack ground-truth knowledge about storage collision issues in these randomly selected contracts, comparing execution traces before and after patching provides limited insights. Therefore, we will further evaluate COLLISIONREPAIR on verified ground-truth contracts.

6.3 Effectiveness

We further apply COLLISIONREPAIR to the 913 confirmed vulnerable contracts (12,526 proxy-logic pairs) obtained from the CRUSH project. For each of these pairs, we use differential testing to compare execution traces before and after


(a) Past Transactions


(b) Targeted Tx w/o Collisions

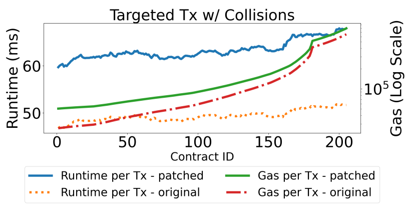

(c) Targeted Tx w/ Collisions

Figure 9: Runtime and Gas Consumption for Vulnerable Contracts

patching. Each test input comprises a sequence of multiple transactions, as storage writes causing collisions often span across separate transactions. In particular, we perform tests under three conditions:

(1) Past Transactions: We begin by replaying all 3,261,133 past transactions in sequences on both the original and patched contracts. To achieve this, we use the same emulation-based approach as in the correctness experiments.

Most past transactions are benign and unlikely to trigger the vulnerabilities. To test the patch, we manually craft transaction sequences that access shared storage and vary their order to generate multiple test cases.

(2) Targeted Transactions without Collisions: We first create 1,000 transaction sequences that trigger vulnerable code without causing storage collision issues, to verify the execution states of the vulnerable and patched code under normal conditions.

(3) Targeted Transactions with Collisions: We then generate 1,000 attack transaction sequences that induce storage collisions, allowing us to examine the execution states of the vulnerable/patched code under attack conditions.

To compare execution traces, we follow Algorithm 2 to verify the equality of each pair of key events along with their input and output states. For efficiency, we first apply a filtering step by comparing the counts of key events, such as `SSTORE` and `CALL`, in both traces. If the counts differ, we can immediately conclude that the traces are not equivalent. Otherwise, we proceed with the algorithm to verify trace equivalency. For traces found to be inequivalent, we conduct a manual inspection to identify the root causes.

Our results show that 13 contracts failed to deploy after patching. We found that these contracts caused storage collisions in their constructors, which run during the deployment transaction. The patch code successfully prevented these collisions, thereby halting the deployment process.

For the remaining contracts, when tested with **(1) Past Transactions**, we observe equivalent execution traces pre- and post-patching, indicating that the monitoring code in the patch did not detect any collisions. Note that while our patch verification checks semantic equivalence based on API call sequences and storage accesses, it may miss subtle high-level behaviors such as auctions or voting. To address this, we manually reviewed 2,200 of 6,018 random samples and found no changes in business logic.

With **(2) Targeted Transactions without Collisions**, all patched contract execution traces aligned with the unpatched versions. This demonstrates that, under normal conditions, our patch code preserves the original contract behavior with ***zero false positives***. In contrast, under the same conditions, static analysis tools like CRUSH or USCHUNT would fully block the use of these contracts, even if they are legitimate.

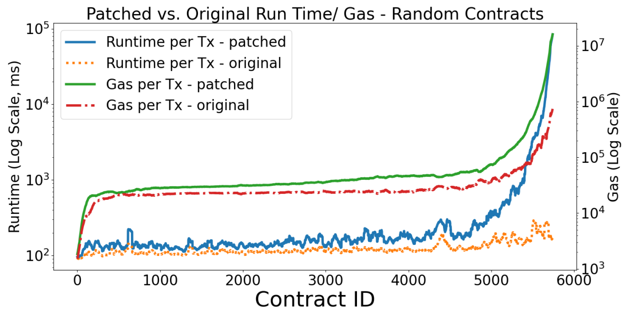


Figure 10: Runtime & Gas Costs for Random Samples

During testing with **(3) Targeted Transactions with Collisions**, every post-patching trace was shorter than its pre-patching counterpart. We confirmed that these differences resulted from precise intervention by our patch code, which successfully detected impending storage collisions and reverted the transaction promptly, with ***zero false negatives***.

Comparison with Existing Tools. Existing tools lack multi-point monitoring, which is essential for addressing stateful bugs. To partially support storage collision patching, we repurpose their reentrancy patchers—originally designed to guard shared lock variables—to block access to specified storage slots. We adapt SMARTSHIELD, ELYSIUM, SGUARD, SMARTFIX to produce stateless patches. EVMPATCH remains unavailable despite our inquiry. Testing on 12,526 vulnerable contracts shows these tools cause 42.6% transaction failures due to context-agnostic blocking, while COLLISIONREPAIR yields no false positives.

6.4 Runtime and Gas Overhead

Next, we measure the additional runtime and gas costs introduced by the patch code. While our patch serves as a first-aid solution to buy time for developers to implement an optimal fix, we aim to ensure its overheads remain reasonable to encourage adoption. We specifically evaluate the overheads in four scenarios using both random samples and vulnerable contracts from CRUSH: *(S1)* 6,018 random samples tested with past transactions, *(S2)* 913 vulnerable contracts (12,526 pairs) tested with past transactions, *(S3)* vulnerable contracts executed with transactions targeting vulnerable code but without collisions, and *(S4)* vulnerable contracts executed with transactions that trigger collisions.

Figure 10 illustrates the runtime and gas costs before and after patching for *(S1)*, while Figure 9 shows the average runtime and gas consumption for each of the 205 contract groups pre- and post-patching for *(S2)*, *(S3)*, and *(S4)*, sorted by the average of the four costs. Table 1 illustrates the average overheads per transaction. As shown, our patch code yields an average runtime overhead of around 19.25%. The runtime overhead for random past transactions is approximately 17%, about 12% lower than for targeted transactions, indicating that functions with storage collision risks generally involve more storage access and require additional monitoring. The average gas overhead is 32.2%, with gas costs for targeted transactions similarly higher.

Discussion. Admittedly, our patch code introduces a moderate overhead. For comparison, EVMPATCH reported an additional gas cost of 255 per transaction for the Useless Ethereum Token [21], equivalent to a 3% gas overhead, though it did not disclose runtime overhead. However, while EVMPATCH only inserts minimal code into specific parts of the contract—e.g., a single line for access control in select functions or an integer overflow check for limited arithmetic operations—our approach instruments every storage access, a common operation in smart contracts. For example, the `approve()` function, frequently called in token contracts, includes `SSTORE`s to update the `allowances` data.

In fact, our instrumentation is comparable to traditional taint tracking or software fault isolation (SFI) techniques. The runtime overhead we observe aligns with theirs, such as the 32% overhead reported for TaintDroid [20] and the 21% protection overhead for efficient SFI [50].

Due to these overheads, COLLISIONREPAIR is most cost-effective when applied to potentially vulnerable contracts flagged by static analysis tools such as CRUSH—blocking attacks while preserving legitimate transactions. The runtime and gas costs can be further reduced through selective instrumentation. For instance, static dataflow analysis for non-interference or manually defined rules could identify storage locations safe from collision attacks and exempt them from monitoring. We leave this exploration to future work.

In contrast, external calls and network communication contribute only about 10% to the overall gas and runtime overheads, making them a less significant source of inefficiency. We argue that our network-based monitoring architecture is justified. In fact, the proxy-logic model used by upgradeable contracts also relies on network communication, inherently making it slower than simple logic contracts. However, this model has become increasingly popular due to its flexibility and upgradability. Similarly, while embedding monitoring code directly into the proxy or logic contract could reduce costs, it would make modifying the monitoring logic significantly more difficult when updates are needed.

Table 1: Average Runtime and Gas Overheads

Scenario	Runtime Overhead	Gas Overhead
Random Samples, Past Tx	16.67%	33.95%
Vulnerable, Past Tx	20.24%	23.82%
Vulnerable, Targeted Tx w/o Collision	30.98%	35.63%
Vulnerable, Targeted Tx w Collision	28.30%	32.46%
All Cases	19.25%	32.18%

6.5 Case Study

Finally, we investigate representative smart contracts to gain insights into the applicability of COLLISIONREPAIR.

Case 1: Token Contracts. Three token contracts—Audius Governance [8], Audius Token [38], and xToken [53]—previously had storage collision issues, which were resolved in later versions. In Table 2, we label these buggy and fixed versions as V1 and V2, respectively. Interestingly, both V1 and V2 contracts follow the EIP-1967 protocol, designed to prevent storage collisions, but the earlier version's implementation was flawed, still leading to this vulnerability.

These vulnerabilities are unique because they exist in the constructors, causing storage collisions during deployment. In our experiments, we observed that the patch code accurately detects the collision, blocking deployment and effectively preventing any subsequent transactions on the vulnerable V1 contracts. This outcome is significant, given the volume of transactions on these buggy versions—for instance, over 137K transactions were executed on the vulnerable Audius Token V1—demonstrating COLLISIONREPAIR's impact by closing this huge attack window.

We also patched the V2 version, where EIP-1967 was correctly implemented. Thus, our patch allowed all transactions to replay successfully, with moderate gas and runtime overheads, averaging around 20.8% and 19.4%, respectively.

Case 2: Decentralized Finance (DeFi). Major DeFi contracts such as Compound III [12] and DerivaDEX [18] contain 1K and 2K lines of code, implementing complex financial exchange logic. Despite this complexity, COLLISIONREPAIR successfully applies patch code to these contracts. Since they strictly follow protocols like EIP-1967 and EIP-2535 to prevent storage collisions, our patch detects no attacks, allowing past transactions to execute correctly with reasonable gas and runtime overheads, as shown in Table 2.

Case 3: Indirect Patch Impact. LoanToken [23] is a contract that includes a `gasleft()` check. This check is used to gracefully revert transactions when low remaining gas is detected. This contract has no storage collision issues. When replaying its historical transactions and comparing traces pre- and post-patching, we observe that the traces are always consistent if we disregard storage writes guarded by the `gasleft()` condition. However, when all `SSTORE` operations are considered, the traces sometimes match and sometimes differ across multiple replays. Further investigation reveals

Table 2: Case Studies for Token and DeFi Contracts

Contract	EIP	Correct Implementation	Txs		Collision Detected	Increased Code		Gas Overhead	Runtime Overhead
			Replayed	All		Proxy	Logic		
Audius Governance V1	EIP-1967	No	0	873	Y	7.2%	3.9%	-	-
Audius Governance V2	EIP-1967	Yes	340	340	N	7.2%	3.9%	22.52%	21.50%
Audius Token V1	EIP-1967	No	0	134,161	Y	7.2%	2.6%	-	-
Audius Token V2	EIP-1967	Yes	137,271	137,271	N	7.2%	3.9%	21.32%	20.10%
xToken V1	EIP-1967	No	0	142	Y	3.3%	3.4%	-	-
xToken V2	EIP-1967	Yes	16	16	N	3.3%	2.6%	18.70%	16.56%
Compound III	EIP-1967	Yes	70,312	70,312	N	4.8%	2.5%	25.60%	22.30%
DerivaDEX	EIP-2535	Yes	6,913	6,913	N	7.3%	3.0%	27.53%	24.42%

this variation is due to fluctuations in remaining gas. Our proposed patch verification process effectively mitigates this implicit impact of gas dynamics, accurately identifying the equivalence of key execution traces.

7 Related Work

Vulnerability Patching for Smart Contracts. Many studies [25,33,37,40,41,48,55,57] propose automated techniques to monitor smart contract runtime behaviors, blocking exploits while allowing legitimate execution. Early approaches [33,40] have limitations, requiring EVM modifications [40] or access to Solidity source code [33, 37, 48, 55], which reduces practicality. More recent work, such as EVMPatch [41], SMARTSHIELD [57], and Elysium [25], directly patches EVM bytecode, showing greater promise. However, these techniques are essentially stateless and cannot address storage collision issues, which are inherently stateful problems.

Security Vetting of Smart Contracts. Many efforts [26, 28,31,32,35,39,40,44,47,49,51] have been made to automatically verify smart contract code to detect security risks. Existing tools detect both syntax-based errors [26, 28, 32, 35, 40] and semantic-level defects [31, 39, 46, 47, 49]. Recent studies [10, 42] further analyze upgradeable contracts to identify the storage collision problems. In comparison, CollisionRepair takes a step further and can effectively mitigate actual storage collision attacks at runtime.

Automated Patching. Efforts to automate patch generation or filter malicious inputs are closely related to our work. AutoPaG [34] analyzes source code to generate patches for out-of-bound exploits, while IntPatch [56] uses dataflow analysis to identify and instrument integer-overflow vulnerabilities with runtime checks. Sidiroglou and Keromytis [45] apply automated patches to vulnerable code, targeting zero-day worms. Newsome et al. [36] propose a filter based on execution traces to block specific attacks, and ShieldGen [16] creates data patches or signatures for unknown vulnerabilities from zero-day attack instances. Razmov and Simon [54] automate filter generation using formal descriptions of application input assumptions.

8 Conclusion

In this paper, we propose CollisionRepair, an automated patching technique for mitigating storage collision risks. CollisionRepair monitors storage access sequences between proxy and logic contracts by defining an "ownership" property for storage locations. It then replays historical transactions to recover existing storage ownership, ensuring the patched code aligns with the current state. A gas impact-aware differential analysis is applied to verify the patch, distinguishing genuine behavioral changes from variations caused by gas usage. Our evaluation on 12,526 real-world contracts shows that CollisionRepair effectively mitigates storage collision attacks without interfering with normal contract operations.

Acknowledgment

We would like to thank the anonymous reviewers for their constructive comments. We also thank the authors of Crush for providing their dataset of vulnerable upgradable smart contracts. This work was supported in part by NSF OAC-2115167, Battelle Energy Alliance, LLC Contract No. 313367, a Stellar Development Foundation gift. Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the funding agency.

Ethics Considerations

In this paper, we presented CollisionRepair, a system designed to patch upgradeable smart contract bytecode with storage collision issues. Our study was conducted entirely in controlled environments using isolated blockchain snapshots. Consequently, traditional ethical concerns commonly associated with cybersecurity research—such as experimentation on live systems, data privacy, and human subject involvement—were not applicable. Our research posed no risk to actual blockchain deployments or real-world operations.

CollisionRepair is a mitigation, not detection, tool—its patch code reveals vulnerabilities only when attacks occur. The vulnerable contracts in our study were provided by the Crush authors, who had already reported the issues and observed attacks. Transaction replays confirm that CollisionRepair blocks these known attacks. No new attacks were found in additional samples; any future discoveries will be reported to CISA and developers based upon their information available on Etherscan.

This work adheres to all relevant legal and ethical standards, and no additional regulatory approvals were required. Our contributions advanced the field of blockchain and smart contract security without introducing any new risks or ethical challenges.

Open Science

All code, documentation, and experimental data are publicly available at: https://github.com/scpatch/Storage-Collision-Patch. The repository includes the source code of CollisionRepair, sample contracts (both original and patched), and raw experimental results.

References

[1] Octopus: Security Analysis tool for Blockchain Smart Contracts. https://github.com/quoscient/octopus/, 2020.

[2] The Ethereum Blockchain Explorer. https://etherscan.io/, 2021.

[3] Compound, 2022.

[4] Discover, collect, and sell extraordinary NFTs, 2022.

[5] Trade NFTs, Get Rewards, 2022.

[6] UNISWAP PROTOCOL, 2022.

[7] Storage collision, 2024.

[8] AUDIUS. Governance. https://docs.audius.org/learn/contributing/governance/, 2024.

[9] Gabriel Barros and Patrick Gallagher. Erc-1822: Universal upgradeable proxy standard (uups). https://eips.ethereum.org/EIPS/eip-1822, 2019.

[10] William E. Bodell III, Sajad Meisami, and Yue Duan. Proxy hunting: Understanding and characterizing proxy-based upgradeable smart contracts in blockchains. In *32th USENIX Security Symposium (USENIX Security 23)*, 2023.

[11] certik. Diamond proxy contracts: Best practices. https://www.certik.com/resources/blog/7laIe0oZGK6IoYDwn0g2Jp-diamond-proxy-contracts-best-practices, 2023.

[12] Compound. Compound iii. https://docs.compound.finance/, 2024.

[13] ConsenSys. One click blockchain. https://archive.trufflesuite.com/ganache/, 2024.

[14] ConsenSys. Unleash the full potential of web3. https://www.infura.io/, 2024.

[15] Cryptopedia. What was the dao? https://www.gemini.com/cryptopedia/the-dao-hack-makerdao, 2023.

[16] Weidong Cui, Marcus Peinado, Helen J. Wang, and Michael E. Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *2007 IEEE Symposium on Security and Privacy (SP '07)*, 2007.

[17] Michael del Castillo. The dao attacked: Code issue leads to $60 million ether theft. https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft, 2016.

[18] DerivaDEX. The next generation of crypto derivatives. https://www.derivadex.com/, 2024.

[19] EIP. Erc-1967: Proxy storage slots. https://eips.ethereum.org/EIPS/eip-1967, 2024.

[20] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow tracking system for realtime privacy monitoring on smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, October 2010.

[21] Etherscan. Contract 0x27f706edde3ad952ef647dd67e24e38cd0803dd6. https://etherscan.io/address/0x27f706edde3ad952ef647dd67e24e38cd0803dd6, 2024.

[22] Etherscan. Contract 0x687642347a9282be8fd809d8309910a3f984ac5a. https://etherscan.io/address/0x687642347a9282Be8FD809d8309910A3f984Ac5a#code, 2024.

[23] Etherscan. Contract 0x687642347a9282be8fd809d8309910a3f984ac5a. https://etherscan.io/address/0x687642347a9282Be8FD809d8309910A3f984Ac5a#code, 2024.

[24] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, pages 436–454. Springer, 2014.

[25] Christof Ferreira Torres, Hugo Jonker, and Radu State. Elysium: Context-aware bytecode-level patching to automatically heal vulnerable smart contracts. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '22, 2022.

[26] Joel Frank, Cornelius Aschermann, and Thorsten Holz. {ETHBMC}: A bounded model checker for smart contracts. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[27] JAKE FRANKENFIELD. 51% attack. https://www.investopedia.com/terms/1/51-attack.asp, 2019.

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

[29] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 129–144, 2015.

[30] William E Bodell III, Sajad Meisami, and Yue Duan. Proxy hunting: Understanding and characterizing proxy-based upgradeable smart contracts in blockchains. In *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[31] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *Proceedings of the 2018 Network and Distributed System Security Symposium*, 2018.

[32] Johannes Krupp and Christian Rossow. Teether: Gnawing at ethereum to automatically exploit smart contracts. In *Proceedings of the 27th USENIX Conference on Security Symposium (USENIX Security'18)*, 2018.

[33] Ao Li, Jemin Andrew Choi, and Fan Long. Securing smart contract with runtime validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, 2020.

[34] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, Bing Mao, and Li Xie. Autopag: towards automated software patch generation with source code root cause identification and repair. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, ASIACCS '07, 2007.

[35] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*, 2016.

[36] James Newsome, David Brumley, Dawn Song, Jad Chamcham, and Xeno Kovah. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *NDSS*, 2006.

[37] Tai D Nguyen, Long H Pham, and Jun Sun. Sguard: towards fixing vulnerable smart contracts automatically. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1215–1229. IEEE, 2021.

[38] OAF. Introducing $audio, the audius platform token. https://audius.org/en/token, 2024.

[39] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy (Oakland)*, 2020.

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

[41] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. EVMPatch: Timely and automated patching of ethereum smart contracts. In *30th USENIX Security Symposium (USENIX Security 21)*, August 2021.

[42] Nicola Ruaro, Fabio Gritti, Robert McLaughlin, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. Not your type! detecting storage collision vulnerabilities in ethereum smart contracts. In *Proceedings of the 2024 Network and Distributed System Security Symposium)*, 2024.

[43] Rust. Understanding ownership. https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html, 2024.

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

[45] S. Sidiroglou and A.D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security & Privacy*, 2005.

[46] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. Smartest: Effectively hunting vulnerable transaction sequences in smart contracts through language model-guided symbolic execution. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1361–1378. USENIX Association, August 2021.

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

[48] Sunbeom So and Hakjoo Oh. Smartfix: Fixing vulnerable smart contracts by accelerating generate-and-verify repair using statistical models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, 2023.

[49] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, 2018.

[50] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev.*, December 1993.

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

[52] S. Wu, Z. Yu, D. Wang, Y. Zhou, L. Wu, H. Wang, and X. Yuan. Defiranger: Detecting defi price manipulation attacks. *IEEE Transactions on Dependable and Secure Computing*, dec 2023.

[53] xToken. Simple, efficient staking & liquidity strategies. `https://xtoken.market/`, 2024.

[54] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for android applications. In *21st USENIX Security Symposium (USENIX Security 12)*, Bellevue, WA, August 2012.

[55] Xiao Liang Yu, Omar Al-Bataineh, David Lo, and Abhik Roychoudhury. Smart contract repair. *ACM Trans. Softw. Eng. Methodol.*, sep 2020.

[56] Chao Zhang, Tielei Wang, Tao Wei, Yu Chen, and Wei Zou. Intpatch: automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In *Proceedings of the 15th European Conference on Research in Computer Security*, ESORICS'10, 2010.

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

[58] Zhuo Zhang, Zhiqiang Lin, Marcelo Morales, Xiangyu Zhang, and Kaiyuan Zhang. Your exploit is mine: Instantly synthesizing counterattack smart contract. In *32nd USENIX Security Symposium (USENIX Security 23)*, August 2023.

Appendices

A Patched Code for Motivating Example

Figure 11 shows a partial view of the patched code for our motivational example. To enhance clarity and readability, we present the patch code using Solidity source code. Notice that, however, we actually insert the patch code directly into the original bytecode programs.

In particular, at the beginning of the `constructor()`, we insert the function calls to `initOwnership()` (ln.4) and `initCallstackMap()` (ln.5), which can therefore establish the initial states at deploy time. The ownership status may not be empty, as the contract may have operated for some time. In that case, we will need to recover the current ownership state via replaying past transactions (discussed in Section 5).

Then, at the start of each function, we introduce code to verify whether an non-empty call stack already exists for the current transaction caller, `msg.sender` (ln.6,14,25). If not, that means this is the starting point of this transaction, where we must initialize the `msg.sender`'s call stack via pushing the entry point function's host contract and its storage onto the stack. For instance, in the `constructor()`, such information is a pair: ⟨"`Proxy`","`Proxy`"⟩. Additionally, before each function call, we insert instructions to update the corresponding call stack. For example, when `LOGIC.initialize()` is invoked using a delegate call, we push the new contract `Logic` and the storage it uses, `Proxy`'s storage, onto the stack (ln.9). Furthermore, at the end of each function, before function returns, we restore the call stack (ln.11,18,33).

To detect storage collisions, we add code before each memory read or write to perform a check (ln.16,27,29,31). For instance, to ensure `fees[(uint) t] = percent` does not cause a collision, we pass the current context at the top of the call stack (`peekCallstack(getCallstack(msg.sender))`), the accessed memory address (`t*8`), and the accessed data size (8)

```
contract Proxy {
    [..]
    constructor() {
        initOwnership();
        initCallstackMap();
        if !existNonEmptyCallstack(msg.sender){
            pushCallstack(newCallstack(msg.sender), <"
                 Proxy", "Proxy">);}
        [..]
        pushCallstack(getCallstack(msg.sender), <"
             Logic","Proxy">);
        LOGIC.initialize();
        popCallstack(getCallstack(msg.sender));
    }
    function updateFee(FeeType t, uint64 percent) {
        if !existNonEmptyCallstack(msg.sender){
            pushCallstack(newCallstack(msg.sender), <"
                 Proxy", "Proxy">);}
        checkEditOwner(peekCallstack(getCallstack(msg.
             sender)),t*8,8);
        fees[(uint) t] = percent;
        popCallstack(getCallstack(msg.sender));
    }
}

contract Logic is Initializable {
    [..]
    function initialize() external {
        if !existNonEmptyCallstack(msg.sender){
            pushCallstack(newCallstack(msg.sender), <"
                 Logic", "Logic">);}
        checkOwner(peekCallstack(getCallstack(msg.
             sender)),0,1);
        require(!initialized);
        checkEditOwner(peekCallstack(getCallstack(msg.
             sender)),0,1);
        initialized = 1;
        checkEditOwner(peekCallstack(getCallstack(msg.
             sender)),32*257,20);
        admin = msg.sender;
        popCallstack(getCallstack(msg.sender));
    }
}
```

Figure 11: Partial View of the Patched Contract

to the function `checkEditOwner()`. This function leverages this information to check the ownership status, and either revert the transaction or update the ownership state, depending on whether a collision will occur.

B Bytecode Patch for Monitoring `SSTORE`

Figure 12 presents our complete bytecode patch for monitoring `SSTORE` operations, with inlined comments explaining most of the instructions.

```
...                         // Relocated instructions from top
1215: PUSH8 0xFFFFFFFFFFFFFFFF  // Relocated instruction
1224: DUP1                  // Duplicate the bitmask (0xF...F)
1225: PUSH1 0x40            // Push the free memory pointer
1227: MLOAD                 // Load the base address from 0x40
1228: PUSH1 0x24            // Push the offset 0x24
1230: ADD                   // Add the offset to base address
1231: MSTORE                // Store the mask as the 2nd arg
1232: DUP3                  // Duplicate the 2^64
1233: PUSH1 0x40            // Push the free memory pointer
1235: MLOAD                 // Load the base address from 0x40
1236: PUSH1 0x44            // Push the offset 0x44
1238: ADD                   // Add the offset to base address
1239: MSTORE                // Store 2^64 as the 3rd arg
1240: PUSH4 0x56B10083// Selector of ``check()'' function
1245: PUSH1 0xE0            // Push 0xE0 (used for shifting to
                            // create a method ID)
1247: SHL                   // Shift the function selector left
                            // by 0xE0 (224 bits)
1248: PUSH1 0x40            // Push the free memory pointer
1250: MLOAD                 // Load the base address from 0x40
1251: MSTORE                // Store the shifted function
                            // selector getStoreValue(uint256
                            // slot, bytes32 offset, bytes32
                            // data)
1252: AND                   // Relocated instruction
1253: MUL                   // Relocated instruction
1254: OR                    // Relocated instruction
1255: SWAP1                 // Relocated instruction
1256: DUP1                  // Duplicate the slot number
1257: PUSH1 0x40            // Push the free memory pointer
1259: MLOAD                 // Load the base address from 0x40
1260: PUSH1 0x4             // Push the offset 0x4
1262: ADD                   // Add the offset to base address
1263: MSTORE                // Store slot number as 1st arg
1264: PUSH1 0x0             // Return value size
1266: PUSH1 0x0             // Return value offset
1268: PUSH1 0x64            // Argument size
1270: PUSH1 0x40            // Push the memory pointer
1272: MLOAD                 // Push the memory pointer content
                            // as argument offset
1273: PUSH1 0x0             // Push the value (0 wei)
1275: PUSH20 0xFE..1EE// The monitoring contract address
1296: GAS                   // The remaining gas
1297: CALL                  // Call the ``check()'' function
1298: SSTORE                // If check passes, execute SSTORE
1299: POP                   // Pop the checker's return value
1300: POP                   // Relocated instruction
1301: POP                   // Relocated instruction
1302: POP                   // Relocated instruction
1303: JUMP                  // JUMP to the next basic block
```

Figure 12: More Complete Patch Code to Monitor `SSTORE`