Uncovering Hidden Proxy Smart Contracts for Finding Collision Vulnerabilities in Ethereum

Cheng-Kang Chen**, Wen-Yi Chu**, Muoi Tran†, Laurent Vanbever‡, Hsu-Chun Hsiao*
*National Taiwan University, {r10922187, r09922070, hchsiao}@csie.ntu.edu.tw
†Chalmers University of Technology and University of Gothenburg, muoi@chalmers.se
‡ETH Zürich, lvanbever@ethz.ch

***Abstract*—The proxy design pattern allows Ethereum smart contracts to be simultaneously immutable and upgradeable, in which an original contract is split into a proxy contract containing the data storage and a logic contract containing the implementation logic. This architecture is known to have security issues, namely function collisions and storage collisions between the proxy and logic contracts, and has been exploited in real-world incidents to steal users' millions of dollars worth of digital assets. In response to this concern, several previous works have sought to identify proxy contracts in Ethereum and detect their collisions. However, they all fell short due to their limited coverage, often restricting analysis to only contracts with available source code or past transactions.**

To bridge this gap, we present PROXION, an automated cross-contract analyzer that identifies all proxy smart contracts and their collisions in Ethereum. What sets PROXION apart is its ability to analyze *hidden* smart contracts that lack both source code and past transactions. Equipped with various techniques to enhance efficiency and accuracy, PROXION outperforms the state-of-the-art tools, notably identifying millions more proxy contracts and thousands of unreported collisions. We apply PROXION to analyze over 36 million alive contracts from 2015 to 2023, revealing that 54.2% of them are proxy contracts, and about 1.5 million contracts exhibit at least one collision issue.

I. INTRODUCTION

Ethereum is a popular blockchain that enables decentralized applications on the Internet, such as decentralized finance, voting systems, and non-fungible token marketplaces, through the creation and execution of smart contracts. These smart contracts are deployed onto the Ethereum blockchain network (i.e., replicated across all participating nodes), facilitating autonomous and trustless execution of the contract's functions. Smart contracts are immutable — once deployed, they cannot be changed or tampered with. While immutability ensures the integrity and reliability of smart contract execution, it poses challenges for updating smart contracts (e.g., to fix bugs or introduce new features) since existing states (e.g., stored data, balances) must be migrated to new smart contracts.

To enable smart contracts' upgradeability while still adhering to their immutability, the *proxy design pattern* has recently emerged in several Ethereum Improvement Proposals (EIPs) (e.g., [7], [8], [9], [10], [11]) and in major blockchain-based companies [12], [13]. Under this pattern, an original smart contract is decoupled into two contracts: a *proxy contract* that contains the data storage and a *logic contract* that contains the implementation logic. The two contracts interact via delegate calls that allow the logic contract's functions to be executed in the context of the proxy contract's storage. To update the implementation of smart contracts under this scheme, developers simply deploy a new logic contract and change the logic contract's address stored in the proxy contract accordingly.

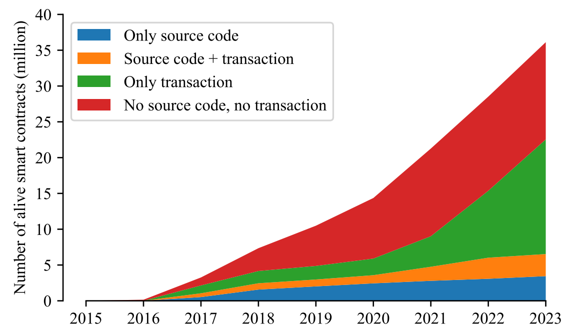


Fig. 1: The accumulated number of alive Ethereum smart contracts till 31 October 2023. Prior works only cover about 18% of smart contracts with source code (blue and orange) [5] or 53% of smart contracts with past transactions (orange and green) [6], while PROXION also applies to the hidden contracts without source code and past transactions (red).

The emerging proxy architecture also comes with new security issues, of which *function collisions* and *storage collisions* are the most notable ones. Particularly, smart contract developers may deliberately or accidentally create conflicts in the storage layouts or function identifiers between the proxy and logic contracts. When users execute these colliding contracts, such conflicts can lead to stored data and functions being incorrectly accessed. Worse, they can also be exploited by sophisticated adversaries to steal assets from the victims who wrongfully execute malicious functions or data. For example, attackers can create malicious contracts with function collisions that disguise their scamming functionalities, often known as honeypot contracts [14]. Outside of the academic realm, adversaries have leveraged storage collisions to overwrite the owner of Audius contracts, stealing more than a million worth of tokens in the process [15]. Also, a bounty hunter discovered storage collisions in the contract connecting the Ethereum and Arbitrum blockchains, which could potentially be exploited to compromise funds exceeding 250 million dollars [16].

* Both authors contributed equally to this research.

	Smart contract coverage				Collision coverage			
	With source code		Without source code		With source code		Without source code	
	With tx	Without tx	With tx	Without tx	Function	Storage	Function	Storage
EtherScan [1]	✓	✓						
Slither [2]	✓	✓			✓	✓		
Salehi et al. [3]	✓		✓					
USCDetector [4]	✓		✓					
USCHunt [5]	✓	✓			✓	✓		
CRUSH [6]	✓		✓			✓		✓
Proxion (this work)	✓	✓	✓	✓	✓	✓	✓	✓

TABLE I: PROXION uncovers more proxy smart contracts than previous works, especially including the *hidden contracts* that do not have source code and past transactions (acronym: tx) available. As a result, PROXION also discovers more collision vulnerabilities, notably function collisions in contracts lacking source code. PROXION's novel coverage is highlighted in green.

More worryingly, the current best practices of proxy smart contracts fall short for both users and developers. Specifically, users must manually review the source code of both proxy and logic contracts before any interaction, such as sending a transaction. Should a proxy contract lack available source code, it is advised that users avoid engaging with it to prevent potential misuse despite possibly missing out on legitimate services. Developers of proxy smart contracts are recently equipped with the new Transparent Upgradeable Proxy design proposed by OpenZeppelin [17] that minimizes the impacts of function collisions. However, adopting a new proxy design is time-consuming, while existing proxy contracts may already contain vulnerabilities due to human errors.

To facilitate the safety assessment of proxy smart contracts, various systems have been recently introduced to automatically identify collision vulnerabilities [2], [5], [6]. At the high level, these systems typically involve two main phases: (1) identifying proxy contracts and their corresponding logic contracts from historical data, and (2) examining each contract pair to determine if there are any colliding functions or storage slots. While the approach seems straightforward, several challenges remain, leaving a non-negligible number of proxy smart contracts unchecked and potentially vulnerable to exploitation.

First, the source code of smart contracts may be unavailable because the public blockchain contains only their runtime bytecode. This bytecode is not human-readable, making the analysis (e.g., checking for delegate calls in the fallback function) difficult. Indeed, existing tools [5], [2] can only analyze smart contracts when their source code is published (e.g., on EtherScan [1]). Figure 1 shows the accumulated number of *alive*[1] Ethereum smart contracts from 2015 to 2023. Unfortunately, we notice that smart contracts with source code available only account for about 18% of all contracts.

Second, uncovering proxy smart contracts through their historical interactions with other smart contracts may also not always be feasible. Specifically, existing tools analyze all blockchain transactions to detect `DELEGATECALL` instructions, identifying contracts involved as proxy and logic contract pairs [6], [3]. This approach applies to all smart contracts, yet it can result in numerous false positives, as common library calls may also contain such instructions. Furthermore, many smart contracts have never interacted with others, such as those freshly deployed on the blockchain. In fact, our data in Figure 1 indicates that only about 53% of the active smart contracts have had interactions with other contracts and are therefore checkable by these tools.

Failing to cover all smart contracts for proxy detection can lead to undesirable outcomes. A clear consequence is that the scope of current tools in identifying collision problems is also limited. Indeed, no prior research has successfully detected function collisions using only the bytecode of proxy and logic contracts. For example, USCHunt [5] and Slither [2] are restricted to detecting collisions in contracts with accessible source code, whereas CRUSH [6] is tailored to detect only storage collisions. Worse, adversaries might deploy malicious contracts (e.g., honeypot contracts [14]) and hide them from these analysis tools by not publishing their source code or interacting with other contracts. These shortcomings highlight the necessity for a novel, effective assessment method for all contracts, thereby enabling thorough collision checks.

To that end, we introduce PROXION, an automated cross-contract analyzer that aims to uncover all proxy smart contracts in Ethereum. In essence, PROXION emulates the execution of the contract under test with carefully crafted inputs that trigger distinct behaviors of proxy contracts (e.g., making delegate calls to other contracts). Therefore, PROXION does not require the contract's source code or its past transactions like prior works. Table I compares PROXION with related works, highlighting its novel ability to cover hidden contracts lacking source code and historical transactions, as well as function collisions when contract source codes are unavailable.

In addition to providing broader coverage of smart contracts and collisions compared to previous studies, PROXION is also efficient, effective, and accurate. Specifically, PROXION can analyze all 36 million active smart contracts in just 65 hours, processing an average of about 150 contracts per second. Moreover, PROXION has identified thousands of vulnerable contracts with unreported collision issues, affecting at least 11 entities in control of a total of 8 billion USD in stakes. In terms of accuracy, PROXION achieves 78.2% in detecting storage collisions and 99.5% in detecting function collisions, surpassing the performance of state-of-the-art tools. Last but not least, we capture and present several insightful trends in developing proxy contracts over the years.

[1] We exclude the destroyed smart contracts.

II. BACKGROUND

In this section, we provide the background by introducing the Ethereum blockchain (§II-A), reviewing proxy smart contracts (§II-B), and describing their collision issues (§II-C).

A. Ethereum Blockchain

Ethereum operates atop a peer-to-peer network of nodes that collaboratively maintain a global blockchain state consisting of users' accounts and balances. A node consists of an execution client responsible for propagating and executing transactions within the Ethereum Virtual Machine (EVM). The transactions either deploy smart contracts onto the network or execute functions within the deployed contracts. Here, deploying a smart contract means compiling a program written in Solidity [18] or Viper [19] into EVM bytecode and creating a new contract account containing the compiled bytecode and its data storage. Also, executing a contract's functions means users or other contracts send transactions containing input data that meets predefined conditions, namely *call data*, to its account. Specifically, the call data encodes a series of bytes, including a function selector (i.e., signature) followed by the function's arguments. Note that the function selector is the first 4 bytes of the `Keccak-256` hash (e.g., `0xdf4a3106`) of the function's prototype string (e.g., `free_ether_withdrawal()`). If the selector matches no function in the contract bytecode and a *fallback function* exists, the EVMs will execute it instead.

B. Proxy Smart Contracts in Ethereum

Once deployed, smart contracts are immutable, and yet they require updates to introduce new features, correct errors, or address security flaws just like conventional programs. Naively migrating a contract's states and balances into a new account for all updates is not scale, as affected users need to update their workflows to use a new contract address. A more favorable solution to smart contract upgradability is splitting it into a *proxy* contract storing the data and a *logic* contract storing the implementation logic [7], [8], [9], [10], [11].

We illustrate the essence of proxy smart contracts in Figure 2. The proxy smart contract facilitates a delegate call to a function in the logic contract. Here, the delegate call allows the execution of the logic contract's function in the context of the proxy contract's storage. In particular, the user encodes the call data for this function in a transaction sent to the proxy contract. This call data contains a function selector that does not match any existing proxy contract functions, allowing it to be passed to the proxy contract's fallback function. As a result, the delegate call in the fallback function is triggered, executing the logic contract's functions while accessing the proxy contract's storage. To enable upgradeability, the proxy contract stores the logic contract's address, usually in one of its storage slots, see `logic` variable in Figure 2. When upgrading to a new logic contract (e.g., v2 in Figure 2), the user only needs to replace the logic contract's address stored in the proxy contract (e.g., using a `setter` call).

Focusing on contracts with upgradability, we exclude library contracts with reusable code for general use, such as

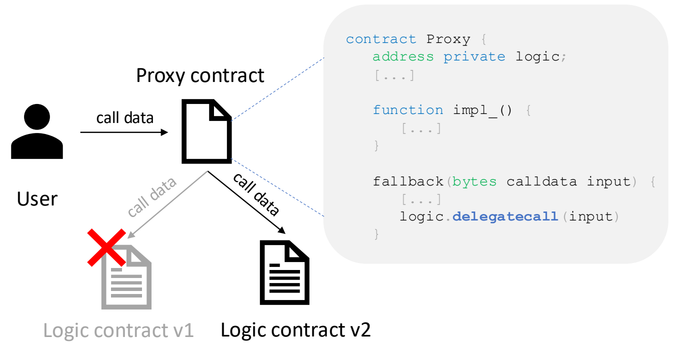


Fig. 2: An example of proxy and logic smart contracts. The proxy contract's delegate call forwards the call data to the upgraded logic contract.

SafeMath [20], from the definition of logic smart contracts. Indeed, when a smart contract calls such libraries, delegate calls occur at different code locations, not in its fallback function, preventing users from choosing whether to execute the library contract's functions. Moreover, all EIPs concerning the proxy smart contract pattern exclude library contracts from their scope. In short, we consider a contract to be a proxy contract if it uses the delegate call in its fallback function to forward the call data it has received to another contract, and any contract receiving forwarded call data from a proxy contract to be a logic contract.

C. Collision Vulnerabilities

Separating data storage and implementation logic in proxy architecture leads to various issues, notably function and storage collisions. These collisions have posed significant risks to proxy and logic contracts, potentially enabling attacks that aim to steal stored assets.

Function collision vulnerability. A *function collision* occurs when a proxy contract's function has the same signature as a logic contract's function. When it happens, users cannot execute the collided functions in the logic contract because the call data is not passed to the fallback function in the proxy contract. The most obvious scenario of function collisions is when a proxy contract's function has the same name as a logic contract's function. The two collided functions may also differ if they share the same first 4 bytes in their hashes. Listing 1 illustrates an example of function collisions, in which two functions `impl_LUsXCWD2AKCc()` (line 11) and `free_ether_withdrawal()` (line 27) have the same 4-byte signature of `0xdf4a3106`. As a result, when the user encodes this signature in the call data, the proxy contract will execute `impl_LUsXCWD2AKCc()` instead of calling the logic contract's `free_ether_withdrawal()`.

Potential exploits. Malicious contract developers can exploit function collisions to trick users into executing honeypot contracts [14]. In such attacks, the adversary creates a logic contract with an enticing function, such as transferring cryptocurrencies to the function caller, which actually collides with a proxy contract's function that steals funds from the caller.

```
1  contract Proxy {
2      address private owner;
3      address private logic; // Logic contract's address
4      address constant USDT = 0xdAC17F958D2ee523a2206206994597...;
5
6      constructor(address impl) {
7          owner = msg.sender;
8          logic = impl;
9      }
10
11     function impl_LUsXCWD2AKCc() public {
12         // a malicious function stealing 1000 USDT from the caller
13         USDT.delegatecall(abi.encodeWithSignature(
14             "transfer(address,uint256)", owner, 1000
15         ));
16     }
17
18     fallback(bytes calldata input) external
19              returns (bytes memory) {
20         (bool success, bytes memory output) =
21             logic.delegatecall(input);
22         return output;
23     }
24 }
25
26 contract Logic {
27     function free_ether_withdrawal() public {
28         // an attractive function that sends the caller 10 Ethers
29         payable(msg.sender).transfer(10 ether);
30     }
31 }
```

Listing 1: A function collision occurs between proxy contract's `impl_LUsXCWD2AKCc()` function and function `free_ether_withdrawal()` in the logic contract due to their same signatures.

```
1  contract Proxy {
2      address private owner; // Storage slot [0x0]
3      [...]
4      address private logic; // Logic contract's address
5
6      constructor(address impl) {
7          logic = impl;
8      }
9      [...]
10     fallback(bytes calldata input) external
11              returns (bytes memory) {
12         (bool success, bytes memory output) =
13             logic.delegatecall(input);
14         return output;
15     }
16 }
17
18 contract Logic {
19     bool private initialized; // Storage slot [0x0]
20     bool private initializing; // Storage slot [0x0]
21
22     function initialize() external {
23         require(initializing || !initialized);
24         initialized = true;
25         initializing = false;
26         owner = msg.sender;
27     }
28     [...]
29 }
```

Listing 2: An example of storage collisions between the proxy and logic contracts. The storage collision occurs at slot 0 between the `owner` variable (20 bytes) in the proxy contract versus `initialized` and `initializing` variables (1 byte each) in the logic contract.

We exemplify the honeypot contracts in Listing 1, where function `free_ether_withdrawal()` in the logic contract allows the caller to withdraw 10 ETH from the contract's balance (line 29). However, since this function has the same function selector with `impl_LUsXCWD2AKCc()`, the user transfers 1,000 USDT to the contract owner instead (lines 13–15). Crafting functions that follow a certain naming pattern or share the same signature with existing functions is relatively easy: we found a function with the same signature as `free_ether_withdrawal()` after approximately 600 million attempts in 1.5 hours with only a commodity laptop.

Storage collision vulnerability. A *storage collision* happens when two variables with different types or interpretations are assigned to the same storage slots across proxy and logic contracts. Because the two contracts share the same storage layout, storage collisions often come from mismatched orders of variable declarations, leading to incorrectly read or overwritten data. The most common cause of storage collisions is when one contract writes to a slot, and another reads from that slot with a different interpretation. Upgrading the logic contract to newer versions that change the order or types of variables also creates storage collisions.

We show an example of storage collisions in Listing 2 where proxy contract's `owner` variable (20 bytes) and logic contract's `initialized` and `initializing` variables (1 byte each) use the same slot 0. Note that if multiple contiguous variables that require less than 32 bytes (for instance, in this case, two `bool` variables are 2 bytes in total) exist, they will be packed into a single storage slot. Also in this example, when users execute a logic contract function involving the `initialized` variable, it may access 1 byte of the `owner` variable in the proxy contract.

Potential exploits. Storage collisions can be exploited to seize control of vulnerable contracts by overwriting the owner's address with that of the attacker [15]. Additionally, adversaries can deceive users into executing malicious logic contracts in which variables have harmless names but are designed to access storage slots in the proxy contract, leading to actions that differ from the user's expectations.

Listing 2 illustrates the vulnerable proxy and logic contracts exploited in the real-world attacks against the Audius cryptocurrency [15]. In particular, the logic smart contract contains an `initialize()` function that sets the transaction's sender as the owner of the contract (line 26) if the owner has not been set previously (line 23). This function is intended to be called only once during contract deployment. However, the `owner` variable in the proxy contract (line 2) and the `initialized` and `initializing` variables in the logic contract are both allocated to the same storage slot number 0. Consequently, even after the `initialized` and `initializing` variables are updated (lines 24–25), indicating the owner has been assigned, the storage slot is immediately overwritten by the new `owner` value (line 26) in the proxy contract. Consequently, the `initializing` variable in the logic contract is always `true`, wrongly indicating that the contracts have not completed the initialization. This allows the `initialize()` function to be executed multiple times and the `owner` variable to be reassigned. Attackers indeed exploited this vulnerability to take control of the Audius governance contracts, as detailed in Audius's post-mortem report [15].

III. Proxion

A. Overview

In this paper, we propose Proxion, an automation tool that aims to reveal proxy smart contracts and their corresponding logic contracts. The main novelty of Proxion is its capability to identify the *hidden proxy smart contracts* that lack both source code and previous transactions.

To uncover proxy contracts, Proxion employs dynamic analysis to verify whether the delegate calls forward the transaction call data in the fallback function. Specifically, for a given smart contract, Proxion disassembles it into opcodes (§III-B) and then emulates their EVM execution using carefully crafted call data (§III-C). If a smart contract is indeed a proxy, a `DELEGATECALL` instruction will appear in the EVM stack and vice versa. Through the emulation of EVM execution, Proxion can also identify the storage locations of the logic contracts' addresses, enabling their easy retrieval from historical blockchain data (§III-D).

After that, any identified pair of proxy and logic smart contracts is further analyzed for collisions. Here, another innovation of Proxion lies in detecting function collisions, even when one or both contracts do not have available source code (§III-E). Specifically, when a contract exists solely in bytecode, Proxion examines the disassembled opcodes to identify the jump instructions corresponding to code blocks of functions. Proxion then extracts the 4-byte data of the function signature that precedes these jump instructions. While the exact function names remain undisclosed, retrieving these signatures is sufficient for Proxion to cross-reference and detect any function collision. We note that existing tools can detect the rest of the collisions. For example, CRUSH [6] can identify exploitable contracts with storage collisions. Slither [2] also can detect function collisions when both proxy and logic contracts have their source code available.

A prototype of Proxion, featuring the proxy smart contract finder and the collision detector, is accessible at *https://github.com/Proxion-anonymous/Proxion*.

B. Disassembling Smart Contracts

We illustrate the two steps of checking if a given smart contract is a proxy contract in Figure 3. In the first step, Proxion determines the tested smart contract is not a proxy if its bytecode does not contain the `DELEGATECALL` opcode, which is the defining factor of all proxy smart contracts. To learn the opcodes of a smart contract, Proxion disassembles its bytecode, which results in a sequence of assembly representation known as opcodes and operands (e.g., [21], [22]).

We implement this disassembler component of Proxion based on Octopus, an open-source security analysis framework that is already capable of translating contract bytecode into certain opcodes and operands [23]. Moreover, we extend Octopus so that it covers recently introduced opcodes in Ethereum, such as `CALL`, `DELEGATECALL`, `CREATE`, and `CREATE2`. This is easily achievable since the opcodes have fixed corresponding bytes. Thereafter, Proxion spots if any

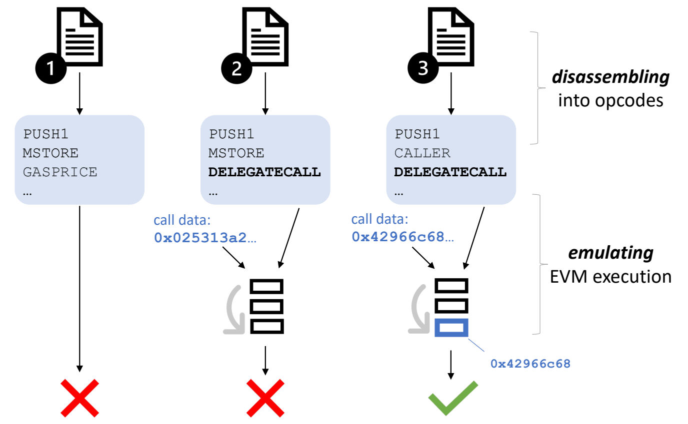


Fig. 3: Proxion identifies proxy smart contracts in two steps. §III-B: the contract's bytecode is disassembled into opcodes. Contracts without a `DELEGATECALL` opcode (e.g., ❶) are not proxies. §III-C: the contract is executed in an EVM emulator with carefully created call data. If this data is not forwarded to another contract, the contract is not a proxy (e.g., ❷) and vice versa (e.g., ❸).

`DELEGATECALL` opcode exists, concluding the smart contract is not a proxy (e.g., contract ❶ in Figure 3) or proceeding to the next step (e.g., contracts ❷ and ❸).

C. Emulating EVM Execution

In the second phase, Proxion checks if delegate calls are triggered in the tested smart contract's fallback function, and they indeed forward the transaction call data to another smart contract. To do so, Proxion triggers the tested smart contract with an emulated EVM and carefully crafted call data. Particularly, this call data contains a random function signature (i.e., with 4 bytes in size) that is *different* from signatures of all other functions in the proxy contract. Thus, it enables the execution of the proxy contract's fallback function. To learn the potentially existing functions' signatures, Proxion identifies the locations of `PUSH4` opcodes in the contract's bytecode and extracts the 4-byte data following each of them. This approach is based on an observation that popular contract compilers (e.g., Solidity, Vyper) always include the function signatures following `PUSH4` opcodes. While not all 4-byte data following `PUSH4` opcodes is a function signature, Proxion safely avoids all of them. Next, Proxion emulates the EVM execution of the tested smart contract along with the generated transaction call data and observes the memory, stack, and storage of each instruction. If Proxion does not observe this data is passed to the logic contract's context after the execution of the `DELEGATECALL` instruction, Proxion marks the tested smart contract as not a proxy (e.g., contract ❷ in Figure 3). Otherwise, the tested smart contract is a proxy (e.g., contract ❸).

To implement our EVM emulator, we extend Octopus [23] to handle opcodes that have values depending on the state of the blockchain. Specifically, we use the values from the latest

block on the blockchain to support the `NUMBER` opcode that pushes the current block's number to the EVM stack. Similarly, we use the values in the latest block for the `BLOCKHASH`, `DIFFICULTY`, `GASLIMIT`, `TIMESTAMP`, and `GASPRICE` opcodes. We also assign fixed values for a few other opcodes, such as `CHAINID`, `BASEFEE`, and `COINBASE`, using the most probable values (e.g., the chain ID of Ethereum's mainnet is 1). Adding these blockchain-related opcodes enhances the fidelity of the EVM emulation (e.g., with fewer runtime errors when encountering them).

Moreover, we implement our EVM emulator to support `CALL` and `DELEGATECALL` opcodes that specifically call another contract and obtain the execution results before proceeding. To do so, we create two EVM emulator instances, one for the caller and another for the callee, and copy the results back from the callee to the stack of the caller instance to simulate the function returning.

For the opcodes that place bytecode on Ethereum at a smart-contract address (e.g., `CREATE` and `CREATE2`), we use a fixed address to ensure that we can retrieve the exact address of a newly created contract. If our EVM emulator encounters this fixed address, we treat it like a normal smart contract. This method is acceptable because of the negligible probability of address collision (e.g., only 1 out of 2^{160} in Ethereum).

D. Finding Associated Logic Contracts

Upon identifying a proxy smart contract, PROXION finds its associated logic contracts, which also can be done by looking into the EVM stack when the `DELEGATECALL` instruction is executed. Indeed, the address of the current logic contract is one of the stack inputs following the `DELEGATECALL` instruction.

Next, PROXION also finds all other logic smart contracts associated with the tested proxy contract in the past. If PROXION observes the found logic contract's address is hard-coded in the proxy contract's bytecode, it considers the test proxy contract follows the *minimal proxy pattern* (i.e., EIP-1167). These minimal proxy contracts include no function but only a delegate call in the fallback function and fix the address of the logic contract in the bytecode. Thus, they are lightweight (e.g., their bytecode is less than 100 bytes) and have only one associated logic smart contract throughout history.

If the found logic contract's address is in a proxy contract's storage slot, PROXION employs a binary search to uncover all addresses stored in that slot. Intuitively, reusing old versions of logic contracts (e.g., containing bugs or missing features) is uncommon. Thus, PROXION implements a binary search for blocks in which the value of the storage slot changes. Specifically, the process begins by comparing the storage slot values at the genesis and the latest block using the `getStorageAt` API. If the values match, it indicates no change in the storage slot within this range of blocks. If they differ, the range is divided into two halves, and the process is repeated to identify all the distinct values ever stored in the proxy smart contract under test. We illustrate how PROXION finds the logic contracts' addresses in Algorithm 1.

Algorithm 1 Finding addresses contained in a storage slot.

Require: $\mathcal{PSC}$: The tested proxy smart contract.
h_{lower}, h_{upper}: The lower and upper bounds for the height of considered blocks (e.g., the genesis block and the latest block).
Ensure: $\mathcal{A}$: The set of logic contracts' addresses associated with $\mathcal{PSC}$.

1: **procedure** PARTITIONBLOCKS(h_{lower}, h_{upper})
2: $\mathcal{V}_{lower} \leftarrow$ getStorageAt($\mathcal{PSC}, h_{lower}$)
3: $\mathcal{V}_{upper} \leftarrow$ getStorageAt($\mathcal{PSC}, h_{upper}$)
4: **if** $\mathcal{V}_{lower} = \mathcal{V}_{upper}$ **then** ▷ Storage slot values are the same.
5: **return** $\{V_{lower}\}$
6: **end if**
7: $h_{mid} \leftarrow \lfloor (h_{lower} + h_{upper})/2 \rfloor$ ▷ Binary search.
8: $\mathcal{A}_{lower} \leftarrow$ PARTITIONBLOCKS(h_{lower}, h_{mid})
9: $\mathcal{A}_{upper} \leftarrow$ PARTITIONBLOCKS($h_{mid} + 1, h_{upper}$)
10: $\mathcal{A} \leftarrow \mathcal{A}_{lower} + \mathcal{A}_{upper} - \{\varnothing\}$
11: **return** $\mathcal{A}$
12: **end procedure**

E. Detecting function collisions without source code

When one or both proxy or logic contracts do not have source code, PROXION analyzes their disassembled opcodes to detect function collisions. It is important to remember that function signatures are always preceded by a `PUSH4` opcode while the reverse is not true (i.e., the data following a `PUSH4` opcode can be arbitrary). Therefore, the challenge lies in identifying which 4-byte data subsequent to the `PUSH4` opcodes actually constitutes a function signature.

To achieve this, PROXION begins by identifying the jump instructions (e.g., `JUMP` or `JUMPI` opcodes), which divide the disassembled code into several basic blocks. These code blocks may represent if-else statements, loops, or function calls, which are distinguishable by how the EVM execution reaches them via the jump instructions. In particular, the EVM execution typically jumps to a specific function after a condition involving its function signature is satisfied (e.g., when call data contains that signature). Thus, PROXION searches for a pattern of opcodes containing `PUSH4` (i.e., pushing 4 bytes), `EQ` (i.e., equal), and `*JUMPI` (i.e., conditional jump). PROXION then extracts the 4-byte sequence following the `PUSH4` opcode within these patterns, treating it as the function signature. Once PROXION has gathered function signatures from both the proxy and logic contracts, PROXION cross-checks them pairwise to find the collisions. To speed up the collision detection in a large dataset of contracts (see Section IV-A), PROXION groups contracts based on their bytecode hash, indicating that the contracts are identical despite having different addresses.

In terms of implementation, PROXION utilizes the state-of-the-art decompiler tool Panoramix [24], an integral part of Etherscan, to disassemble the bytecode and identify the code blocks. PROXION then parses Panoramix's outputs to identify the functions and then retrieve their signatures.

IV. EVALUATION

This section evaluates PROXION in three dimensions: the effectiveness in identifying proxy smart contracts (§IV-B); the accuracy in detecting collision issues (§IV-C); and the efficiency of analyzing a large-scale contract data set (§IV-D).

A. Datasets and Methodologies

We execute PROXION with all active smart contracts as the inputs in October 2023. We also compare PROXION with USCHunt [5] and CRUSH [6] using their independent datasets. Particularly, our evaluation uses the following three datasets:

- **[D1]** contains 36,123,714 *active* contracts as of October 2023. We first query all contracts' addresses and deployment blocks from Google BigQuery [25]. Then, we retrieve their bytecode and storage states from a local archive node [26] and their source code from EtherScan [1]. We also assign the source code of a contract to all other contracts with the same bytecode hash.
- **[D2]** is the Smart Contract Sanctuary dataset [27], used to evaluate USCHunt originally. It contains 329,764 smart contracts deployed from 2017 to 2022, with source code available and collected from EtherScan [1].
- **[D3]** is obtained from the authors of CRUSH [6] in August 2024. It comprises 53,580,899 contracts deployed from July 2015 to April 2023. The contracts in this dataset may lack source code, past transactions, or both. Some contracts in this dataset may already be destroyed.

We execute PROXION with *all three datasets* to evaluate its effectiveness in finding proxy contracts. We also execute USCHunt with **[D2]** and CRUSH with **[D3]** to show that PROXION can identify unknown proxy and logic contracts within datasets already examined by these existing tools.

Regarding the accuracy evaluation, we execute PROXION, USCHunt, and CRUSH with the dataset **[D2]** because we can manually examine its contracts to establish ground truth. Unfortunately, there are several thousand cases where the tools report proxy contracts differently (cf. §IV-B), making manual verification exceedingly time-consuming. To conduct a fair comparison with reduced manual effort, we use the tools to further identify collision issues and then omit the examination of proxy contracts marked as collision-free by all tools. Here, we note that (1) PROXION uses CRUSH's storage collision detection, and (2) CRUSH does not detect function collisions. Specifically, USCHunt, CRUSH, and PROXION identify 116, 102, and 55 storage collisions, totaling 206 unique proxy and logic contract pairs. In the case of function collisions, USCHunt identifies 300, while PROXION reports 557, resulting in another 561 unique cases for manual verification.

To evaluate the performance of PROXION, we apply it on the dataset **[D1]**. Specifically, we operate PROXION on a system equipped with Ubuntu 22.04 OS, featuring 12 cores (24 threads) at 3.8 GHz each and 64 GB of RAM.

B. Effectiveness in Identifying Proxy Contracts

PROXION discovers 19,599,317 proxy contracts, which represent 54.2% of all contracts in dataset **[D1]**. Notably, PROXION *uncovers approximately 1.5 million proxy contracts that are hidden*, lacking both source code and past transactions.

Furthermore, PROXION discovers more proxy contracts than USCHunt and CRUSH when testing against their respective datasets. When running with the dataset **[D2]**, we observe that PROXION experiences notably fewer failure cases than USCHunt. Specifically, USCHunt encounters halt due to contract compilation errors (e.g., unknown compiler versions) in about 30% of cases.[2] Meanwhile, PROXION fails to emulate the execution, for instance, due to insufficient values on the EVM stack in only about 1.2% of contracts. *In total,* PROXION *identifies 35,924 proxy contracts, whereas USCHunt detects only 29,023, which is roughly seven thousand fewer.*

When executing the dataset **[D3]**, CRUSH identifies 26.6% of the examined smart contracts, totaling 14,237,696, are identified as proxy contracts. PROXION reports about 1.2 million fewer proxy contracts than CRUSH in this dataset, totaling 13,042,496 proxy smart contracts. This outcome occurs because CRUSH categorizes any contracts that involve `DELEGATECALL` instructions as proxy contracts, including the one making library calls. In contrast, PROXION does not consider this condition when classifying proxy contracts (cf. Section II-B). When excluding those proxy smart contracts, PROXION *uncovers more 1,667,905 proxy contracts than CRUSH does*, none of which have past transactions available.

		TP	FP	TN	FN	Acc.
Storage collision	USCHunt	33	83	79	11	54.4%
	CRUSH	26	76	86	18	54.4%
	PROXION	27	28	134	17	78.2%
Function collision	USCHunt	299	1	0	261	53.3%
	PROXION	557	0	1	3	99.5%

TABLE II: PROXION has higher accuracy than the state-of-the-art tools in detecting storage and function collisions.

C. Accuracy in Detecting Collision Issues

We report the collision detection accuracy in Table II. *Regarding storage collisions,* PROXION *achieves an accuracy of 78.2%*, surpassing both USCHunt and CRUSH, each achieving an accuracy of 54.4%. USCHunt and CRUSH generate more false positives than PROXION, albeit for different reasons. Specifically, USCHunt mistakenly identifies variables with different names in separate contracts as collisions, often overlooking that one variable may serve as storage padding and is not exploitable. Also, using CRUSH's storage collision detection, PROXION naturally has a similar number of true positives to CRUSH (27 versus 26). However, PROXION still has a higher accuracy than CRUSH thanks to its more effective identification of proxy smart contracts in the preceding step, in which PROXION precisely excludes library contracts.

Regarding function collisions, PROXION *achieves an accuracy of 99.5%, with no false positives and only three false negatives.* In contrast, USCHunt has a lower accuracy of 53.3%, with numerous false negatives due to the underlying Slither failing to identify proxy contracts. Here, PROXION also misses three function collisions due to runtime errors when emulating the EVM execution (e.g., insufficient values).

[2]We run USCHunt with the default compiler flags. There may be fewer errors if the compiler versions are provided when compiling each contract.

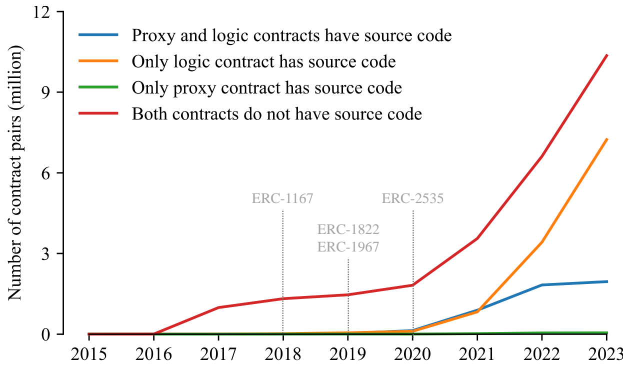


Fig. 4: Accumulated number of pairs of proxy and logic contracts identified by PROXION from 2015 to 2023. In the vast majority of them, the proxy contracts only have bytecode available.

D. Performance

On average, PROXION analyzes a smart contract in just 6.4 milliseconds to determine if it is a proxy, translating to 156.3 contracts per second. This efficiency enables PROXION to process roughly 36 million active contracts in dataset **[D1]** within approximately 65 hours.

Moreover, PROXION's binary search method for identifying logical contracts linked to a proxy contract also significantly boosts its performance. For each proxy contract, PROXION makes 26 `getStorageAt` API calls on average, which is a substantial improvement over the naive method of querying all 15 million Ethereum blocks.

Regarding function collision detection, thanks to PROXION avoiding re-testing identical contracts with the same bytecode, it takes only 6.7 milliseconds to check if a contract pair has a collision issue on average.

V. PROXY SMART CONTRACTS' LANDSCAPE

In this section, we present the comprehensive landscape of proxy smart contracts in Ethereum, resulting from PROXION analyzing all alive contracts for function and storage collisions. Specifically, we present several findings regarding the growth of proxy contracts over the years, the number of found collisions, and the trends in their design patterns and deployments.

First, *over half of the active contracts are proxy or logic contracts, and the majority of proxy contracts do not publish their source code.* We show the number of active proxy contracts identified by PROXION in Figure 4. As of October 2023, there are 19,599,317 proxy contracts, which represent 54.2% of all contracts. Of these, approximately 2 million are pairs of proxy and logic contracts with available source codes for both, as indicated by the blue line. Conversely, about 90% of proxy contracts lack available source codes, as shown by the orange and red lines.

Furthermore, there has been a noticeable divergence in the growth trends of proxy contracts before and after 2020. Only 2 million proxy contracts were deployed before 2020, while 7.6 million proxy contracts were deployed in the first ten months of 2023. These figures closely track the historic adoption of the proxy pattern: the demand for contract upgradeability pre-2018, the testing phase between 2018-2020 with several EIPs [8], [9], [10], [11], and the mainstream phase since 2020 in which more than 93% of contracts are proxy.

Year	Function collisions	Storage collisions
2017	24	0
2018	5,341	7
2019	16,136	37
2020	28,448	34
2021	705,801	725
2022	808,493	2,082
2023	2,541	137
Total	*1,566,784*	*3,022*

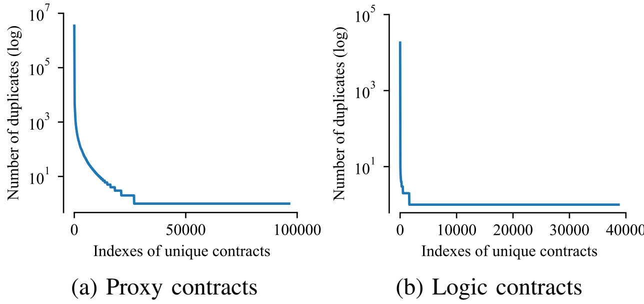
TABLE III: Number of function and storage collisions detected by PROXION. Notably, *1,545,722 (or 98.7%)* proxy contracts with function collisions are actually identical.

Second, PROXION *detects about 1.5 million function collisions, 98.7% of which are duplicated contracts with the same code, and about 3 thousand exploitable storage collisions.* We report the number of function and storage collisions found by PROXION in Table III. Specifically, starting from the 19.5 million pairs of proxy and logic contracts, PROXION detects a total of $1,566,784$ pairs having function collisions and $3,022$ pairs having storage collisions. Notably, 98.7% of the detected function collisions come from many proxy contracts duplicated from the *OwnableDelegateProxy*[3] contract. In those cases, function collisions are caused by the `proxyType()`, `implementation()`, `upgradeabilityOwner()` functions appearing in both proxy and logic contracts, possibly due to contract inheritance [28].

Regarding storage collisions, we identified 91 instances out of 3,022 where both proxy and logic contracts have their source code available. We studied their owners and pinpointed 11 entities, including Ape Finance, Compound, Convex, Curve, GolduckDAO, LeverFi, Poly, Polyhedra Network, Polymath, Tokeny, and Zora. As of this writing, these entities manage stakes totaling 8 billion USD. However, it is important to acknowledge that they may manage stakes in other contracts that are not prone to such vulnerabilities.

Third, *we find the distributions of bytecode uniqueness are heavily skewed, with 42% of proxy contracts duplicating from just three popular contracts.* We highlight the number of unique proxy and logic contracts in Figure 5. Interestingly, while PROXION identifies approximately 19.6 million proxy contracts and 70 thousand associated logic contracts, most of them are actually duplicates (i.e., having the same compiled bytecode) deployed at different addresses. Particularly, Figure 5 reports only $96,420$ and $38,707$ unique proxy and logic contracts, respectively. We see that the distributions of bytecode uniqueness are heavily skewed, in which a small number of contracts are duplicated significantly more than others. To be more specific, the three most popular proxy

[3]https://etherscan.io/address/0x0a08e6058eaaa847a1adb55b0a69b8469ea5a5b3



(a) Proxy contracts (b) Logic contracts

Fig. 5: Most contracts are duplicates: only $96,420$ and $38,707$ unique proxy and logic contracts, respectively. (5a): three proxy contracts are duplicated more than 1 million times. (5b): two logic contracts have the same bytecode with more than ten thousand other contracts.

	# Contracts	Ratio
EIP-1167 [8]	$17,453,264$	89.05%
EIP-1822 [9]	$22,789$	0.12%
EIP-1967 [10]	$196,688$	1.00%
Others	$1,926,576$	9.83%

TABLE IV: The number of proxy contracts following certain design standards. The vast majority of proxy contracts follow the minimal design (EIP-1167).

contracts have more than a million of duplicated contracts, and they are *CoinTool_App*[4], *XENTorrent*[5], and *OwnableDelegateProxy* contracts. We also notice that while most logic contracts are not duplicated frequently, there are two standout logic contracts[6,7] having more than $10,000$ duplicates. We conjecture that these contracts have source code, which renders cloning them uncomplicated and relates to the popularity of the non-fungible token marketplaces in recent years (e.g., *OwnableDelegateProxy* is a core component of the popular Wyvern protocol [29]). It is worth noting that all the duplicates of the popular proxy contracts above associate with the same logic contracts. For example, the *CoinTool_App* logic contract[8] is referenced by almost 3.5 million proxy contracts that are duplicates of the *CoinTool_App* proxy contract. These findings indicate a widespread contract cloning practice that preserves the cloned contract's functionalities. It may be, however, not ideal from the decentralization perspective, as potential bugs or vulnerabilities of the cloned contracts are also propagated, as noted in the previous paragraph or in existing work [30].

Fourth, *the minimal proxy design dominates the standard proxy contracts while a non-negligible portion of non-standard proxy contracts exists.* We present the distribution of proxy contracts' design patterns in Table IV. Notably, most of them (89.05%) adhere to the minimal design standard [8], which includes only the delegate call in the fallback function and

[4]https://etherscan.io/address/0x95a3946104132973b00ec0a2f00f7cc2b67e751f
[5]https://etherscan.io/address/0x4e488a5367daf86cfc71ea3b52ff72ca937efcf8
[6]https://etherscan.io/address/0xf17b1a1f68e1ddaa2e3285437b96ea28af2a2dc0
[7]https://etherscan.io/address/0xa471cd47769c3a788ad9c7b3d8350f195bf672bd
[8]https://etherscan.io/address/0x0de8bf93da2f7eecb3d9169422413a9bef4ef628

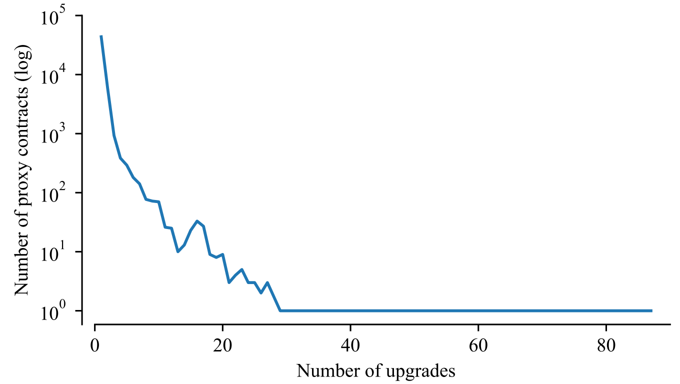


Fig. 6: Number of upgrades for logic contracts in log scale. Most proxy contracts (99.7%) have not upgraded to a newer version of logic contracts.

a hard-coded logic contract address in the bytecode. These minimal proxy contracts are not vulnerable to function and storage collisions due to the absence of variable and function declarations. We further categorize proxy contracts into other standards based on the locations storing their logic contracts' addresses. In particular, $22,789$ contracts are categorized under the EIP-1822 standard (Universal Upgradeable Proxy Standard) as they utilize a specific storage slot derived from the `Keccak-256("PROXIABLE")` hash. Similarly, $196,688$ contracts follow the EIP-1967 standard because they store logic contract addresses in a slot derived from the `Keccak-256` hash of `("eip1967.proxy.implementation")`. We also note that there are 9.83% of the proxy contracts storing their logic contract's addresses in the storage without conforming to any known design patterns.

Fifth, *we find that most proxy contracts have not upgraded to a newer version of logic contracts.* We show the number of contract upgrade events in which the logic contract's address is updated in Figure 6. We find that the number of upgraded smart contracts is insignificant, e.g., only $51,925$ proxy contracts have upgraded their logic implementations throughout history. The majority of these contracts also upgrade only a few times (i.e., having only 1.32 associated logic contracts on average). We also observe that upgrade events are infrequent; only $68,804$ upgrading events ever occurred, meaning, on average, an upgrade happens only once per 200 Ethereum blocks. From the security point of view, such an infrequent upgrade may render proxy contracts less prone to storage collisions, which often arise during contract upgrades.

VI. Discussion

In this section, we discuss a few key points. We first describe the limitations of Proxion and explain why they do not undermine Proxion's contributions (§VI-A). Based on the discussed limitations, we outline a few potential follow-up works for Proxion (§VI-B).

A. Limitations

When analyzing proxy contracts, PROXION misses the contracts following the Diamonds, Multi-Facet Proxy design pattern [11], in which only the function signatures registered by the contract owner can trigger the delegate calls in the fallback function. Unfortunately, PROXION can currently only send randomly generated call data during EVM emulation and, thus, cannot detect these diamond contracts.

Another limitation of the system is the occurrence of runtime errors, which are relatively low (e.g., at 1.2%) during the emulation of EVM execution of smart contracts (cf. Section IV-Bs). Additionally, EVM emulation may inevitably yield results that differ from actual contract execution, although the extent of these discrepancies is not known.

B. Future Work

Future work for PROXION includes identifying proxy contracts that follow the diamond design pattern [11]. A potential solution involves extracting all registered functions from past transactions (similar to CRUSH [6]) and utilizing them to generate call data. When the source code is available, PROXION may employ a static analysis approach like USCHunt [5], combining with the information of the slot storing logic contract's address (i.e., `Keccak-256("diamond.standard.diamond.storage")`).

PROXION can also be extended to analyze proxy smart contracts beyond Ethereum. Similar to USCHunt [5], PROXION may apply to several other blockchains, such as Arbitrum, Avalanche, Binance, Celo, Fantom, Optimism, and Polygon.

VII. RELATED WORK

We consider related work that studies the same target of proxy smart contracts (§VII-A), discusses contract upgradability (§VII-B), or performs analysis on smart contracts (§VII-C).

A. Finding Collisions in Proxy Smart Contracts

Several tools aim to detect proxy smart contracts and their collision issues; see Table I for a comparison.

Slither examines the source code of contracts to determine if they are proxy contracts [2]. However, Slither's proxy detection relies on keyword searches, such as "proxy" or "delegatecall," which may lead to many false positives. Similarly, USCDetector [4] also detects proxy contracts by checking if the contracts' bytecode contains the `DELEGATECALL` opcode and related keywords, such as "update" or "upgrade". Unlike PROXION, Slither does not identify associated logic contracts and USCDetector requires past transactions to do so.

Etherscan is a widely recognized web-based explorer for the Ethereum blockchain, featuring an integrated proxy contract verification tool [1]. This tool identifies contracts with the `DELEGATECALL` opcode as proxy contracts, a result that Etherscan admits may lead to numerous false positives [31]. PROXION applies a similar initial filtering process for proxy contracts and then conducts dynamic analysis, resulting in more precise detection.

USCHunt [5] employs static analysis based on Slither to detect proxy contracts with published source code, and their security vulnerabilities, such as collisions, on eight blockchains, including Ethereum. PROXION focuses on improving the detection of proxy contracts and collision issues, specifically on the Ethereum blockchain, which has shown to be more accurate than USCHunt.

Salehi et al. [3] studied the ownership of upgradability in smart contracts, i.e., finding out who can upgrade the proxy contracts. Similar to PROXION, this work also performs dynamic analysis on smart contracts' bytecode, thus covering more contracts than USCHunt. Unlike PROXION, however, the analysis here involves replaying past transactions to the contracts under tests, thus limiting the effective analysis to only contracts with many transactions.

CRUSH is a newly developed automation tool that detects storage collisions and generates verified exploits [6]. The CRUSH engine is also employed by PROXION to identify storage collisions, particularly for proxy contracts without source code. Like Salehi et al., CRUSH depends on historical transactions to locate proxy contracts, thus missing out on millions of hidden contracts. Additionally, unlike PROXION, CRUSH is not equipped to detect function collisions.

B. Upgradeability in Smart Contracts

The upgradeability in smart contracts has been discussed extensively in several EIPs [8], [9], [10], [11], blog posts [17], [32], and studies [33], [28], [34], [35]. These works focus on designing new proxy patterns to enable upgrading smart contracts at scale or studying the status quo of upgradable contracts. Our work also studies upgradable contracts in Ethereum, which is a subset of proxy smart contracts. We further reveal that upgrading events are actually rare, and functionality cloning is a more popular usage of proxy contracts. Our work, thus, provides an additional discussion to the existing literature on upgradeability in smart contracts.

C. Smart Contracts Analyzers

Many smart contract analyzers have been proposed in recent years to detect smart contract vulnerabilities [36]. Commonly, they can be categorized into static analyzers and dynamic analyzers based on their overall approach. Particularly, static analyzers detect smart contract vulnerabilities by inspecting their source code or bytecode, using techniques such as information flow analysis (e.g., Slither [2], MadMax [37]) or symbolic execution (e.g., Oyente [38], MantiCore [39], Securify [40], teEther [41], Mythril [42], Zeus [43], Osiris [44]). On the other hand, dynamic analyzers execute the test smart contracts and observe the behaviors of vulnerable ones, using fuzzing (e.g., ReGuard [45], ContractFuzzer [46], Confuzzius [47], sFuzz [48], Harvey [49]) or validation (e.g., MAIAN [50], Sereum [51], SODA [52], ESCORT [53]). We refer to a recent survey by Kushwaha et al. [54] for a more comprehensive review of such existing analysis tools. This paper proposes PROXION, a new hybrid contract analyzer focusing specifically on the collision issues of Ethereum smart contracts.

VIII. Conclusion

While the proxy design pattern enables Ethereum smart contract upgrades, it also introduces security risks from function and storage collisions. Previous efforts to detect these collisions fall short due to the limited coverage of testing contracts under test, especially for the hidden ones without source code or transaction history. To fill this gap, we propose PROXION, an automated tool that efficiently uncovers hidden proxy contracts and their collision vulnerabilities. Utilizing PROXION, we analyzed all Ethereum smart contracts and discovered that half are associated with the proxy pattern, and millions are vulnerable to collisions. Observations on proxy smart contracts' evolution suggest an increased adoption of the proxy pattern in the coming years, making PROXION a crucial step toward securing proxy contracts in this anticipated future.

Acknowledgments

We thank the anonymous reviewers and Hsun Lee for their helpful feedback on this paper. This work was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation, the National Science and Technology Council of Taiwan under grants 112-2223-E-002-010-MY4 and 113-2634-F-002-001-MBK, and National Taiwan University under grant 114L7848.

References

[1] EtherScan, "EtherScan," https://etherscan.io/, 2023.
[2] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *IEEE WETSEB*, 2019.
[3] M. Salehi, J. Clark, and M. Mannan, "Not so immutable: Upgradeability of smart contracts on Ethereum," in *WTSC*, 2022.
[4] X. Li, J. Yang, J. Chen, Y. Tang, and X. Gao, "Characterizing ethereum upgradable smart contracts and their security implications," in *Proc. ACM Web Conference*, 2024.
[5] W. E. Bodell III, S. Meisami, and Y. Duan, "Proxy hunting: Understanding and characterizing proxy-based upgradeable smart contracts in blockchains," in *USENIX Security*, 2023.
[6] N. Ruaro, F. Gritti, R. McLaughlin, I. Grishchenko, C. Kruegel, and G. Vigna, "Not your Type! Detecting Storage Collision Vulnerabilities in Ethereum Smart Contracts," in *Proc. NDSS*, 2024.
[7] J. Izquierdo and M. Araoz, "ERC-897: DelegateProxy," 2018.
[8] P. Murray, N. Welch, and J. Messerman, "EIP-1167: Minimal Proxy Contract," 2018.
[9] G. Barros and P. Gallagher, "ERC-1822: Universal Upgradeable Proxy Standard," 2019.
[10] S. Palladino, F. Giordano, and H. Croubois, "ERC-1967: Proxy Storage Slots," 2019.
[11] N. Mudge, "ERC-2535: Diamonds, Multi-Facet Proxy," 2020.
[12] OpenSea, "OpenSea," https://opensea.io/, 2023.
[13] Compound, "Compound," https://compound.finance/, 2023.
[14] C. F. Torres, M. Steichen, and R. State, "The art of the scam: Demystifying honeypots in Ethereum smart contracts," in *USENIX Security*, 2019.
[15] Audius, "How Attackers Stole Around $1.1M Worth of Tokens From Decentralized Music Project Audius," 2022.
[16] M. Truppa, "Arbitrum announces 400 ETH bug bounty payout," 2022.
[17] OpenZeppelin, "Proxy Upgrade Pattern," 2023.
[18] E. Foundation, "The Solidity Contract-Oriented Programming Language," https://github.com/ethereum/solidity, 2024.
[19] V. Buterin, "Viper," https://ethereum-viper.readthedocs.io, 2024.
[20] OpenZeppelin, "Safemath," 2020.
[21] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: thorough, declarative decompilation of smart contracts," in *Proc. IEEE/ACM ICSE*, 2019.
[22] N. Grech, S. Lagouvardos, I. Tsatiris, and Y. Smaragdakis, "Elipmoc: Advanced decompilation of ethereum smart contracts," in *PACMPL*, 2022.
[23] P. Ventuzelo, "Octopus: Security Analysis tool for WebAssembly module and Blockchain Smart Contracts," 2023.
[24] T. Kolinko and Palkeo, "Panoramix - EVM decompiler," 2024.
[25] Google BigQuery, "Ethereum Cryptocurrency," 2023.
[26] Ethereum, "Ethereum Archive Node," https://ethereum.org/en/developers/docs/nodes-and-clients/archive-nodes, 2023.
[27] M. Ortner and S. Eskandari, "Smart Contract Sanctuary," https://github.com/tintinweb/smart-contract-sanctuary, 2023.
[28] G. Zheng, L. Gao, L. Huang, J. Guan, G. Zheng, L. Gao, L. Huang, and J. Guan, "Upgradable contract," *Ethereum Smart Contract Development in Solidity*, 2021.
[29] Wyvern, "Wyvern Protocol," https://wyvernprotocol.com/, 2023.
[30] K. Yan, J. Zhang, X. Liu, W. Diao, and S. Guo, "Bad apples: Understanding the centralized security risks in decentralized ecosystems," in *ACM Web Conference*, 2023.
[31] EtherScan, "What is Proxy Contract?" 2023.
[32] Trail of Bits, "Contract upgrade anti-patterns," 2018.
[33] M. Fröwis and R. Böhme, "Not all code are create2 equal," in *WTSC*, 2022.
[34] P. Klinger, L. Nguyen, and F. Bodendorf, "Upgradeability concept for collaborative blockchain-based business process execution framework," in *Proc. IEEE ICBC*, 2020.
[35] M. Rodler, W. Li, G. O. Karame, and L. Davi, "EVMPatch: Timely and automated patching of Ethereum smart contracts," in *Proc. USENIX Security*, 2021.
[36] N. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts SoK," in *ETAPS*, 2017.
[37] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in Ethereum smart contracts," *Proc. OOPSLA*, 2018.
[38] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM CCS*, 2016.
[39] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *Proc. IEEE/ACM ASE*, 2019.
[40] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM CCS*, 2018.
[41] J. Krupp and C. Rossow, "teEther: Gnawing at Ethereum to automatically exploit smart contracts," in *Proc. USENIX Security*, 2018.
[42] Consensys, "Mythril," https://github.com/ConsenSys/mythril, 2023.
[43] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: analyzing safety of smart contracts." in *Proc. NDSS*, 2018.
[44] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in Ethereum smart contracts," in *Proc. ACSAC*, 2018.
[45] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *Proc. ACM ICSE*, 2018.
[46] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. ACM/IEEE ASE*, 2018.
[47] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts," in *Proc. IEEE EuroS&P*, 2021.
[48] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proc. ACM/IEEE CSE*, 2020.
[49] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *Proc. ACM FSE*, 2020.
[50] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proc. ACSAC*, 2018.
[51] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," in *NDSS*, 2019.
[52] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, and Z. He, "Soda: A generic online detection framework for smart contracts." in *NDSS*, 2020.
[53] C. Sendner, H. Chen, H. Fereidooni, L. Petzi, J. König, J. Stang, A. Dmitrienko, A.-R. Sadeghi, and F. Koushanfar, "Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning." in *Proc. NDSS*, 2023.
[54] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Ethereum smart contract analysis tools: A systematic review," *IEEE Access*, 2022.