An Empirical Study of Proxy Contracts at the Ethereum Ecosystem Scale

Mengya Zhang*†, Preksha Shukla‡†, Wuqi Zhang§†, Zhuo Zhang§¶, Pranav Agrawal‡, Zhiqiang Lin*, Xiangyu Zhang§, Xiaokuan Zhang‡
*The Ohio State University, ‡George Mason University, §Purdue University, ¶Offside Labs

Abstract—**The proxy design pattern separates data and code in smart contracts into proxy and logic contracts. Data resides in proxy contracts, while code is sourced from logic contracts. This pattern allows for flexible smart contract development, enabling upgradeability, extensibility, and code reuse. Despite its popularity and importance, there is currently no systematic study to understand the prevalence, use scenarios, and development pitfalls of proxies. We present the first comprehensive study on Ethereum proxies. To gather a dataset of proxies, we introduce PROXYEX, the first framework to detect proxies from bytecode, achieving over 99% accuracy. Using PROXYEX, we collected a dataset of 2,031,422 Ethereum proxies and conducted the first large-scale empirical study. We analyzed proxy numbers and transaction traffic to understand their current status on Ethereum. We identified four proxy use patterns: upgradeability, extensibility, code-sharing, and code-hiding. We also pinpointed three common issues: proxy-logic storage collision, logic-logic storage collision, and uninitialized contracts, creating checkers for these by replaying historical transactions. Our study reveals that upgradeability isn't the sole reason for proxy adoption in DApps, and many proxies present issues like storage collisions and uninitialized contracts, which enhances the understanding of proxies and guide future smart contract research on the development, usage, quality assurance, and bug detection of proxies.**

I. INTRODUCTION

Ethereum [1], [2] has emerged as one of the leading blockchain platforms with Turing-complete smart contracts, allowing for greater flexibility and functionality in developing Decentralized Applications (DApps). As of February 2024, the total Unique Active Wallets (UAW) on Ethereum reached approximately 152.4 million, with a Total Value Locked (TVL) across all Ethereum DApps at around $7.11 trillion [3].

Data and code of Ethereum smart contracts are closely coupled in the same address. Moreover, the code of a contract is immutable: it cannot be modified once deployed. Such characteristic hinders the flexibility of developing smart contracts and DApps, introducing constraints including restricted code size [4], constrained functionality expansion, challenges in reusing code, and obstacles in patching code, etc [5]. The separation of code and data is deemed to be a preferred practice of software design [6] to improve maintainability, security, understandability, etc, and resolve the aforementioned constraints of DApps.

Fortunately, to mitigate such limitations, the community has proposed the ***proxy*** design pattern to achieve the separation of code and logic. Proxy pattern separates data and logic implementation into two different contracts, namely ***proxy contract*** and ***logic contract***, respectively. Users of the DApp interact and send transactions to the proxy contract, which will use DELEGATECALL, a type of Ethereum Virtual Machine (EVM) Opcode [7], to execute the code of the logic contract on top of the data in the proxy contract, as shown in Fig. 1. With the proxy design pattern, DApps may achieve: 1) *upgradeability and code patching* by replacing the DELEGATECALL target (i.e., the address of the logic contract) with a new one, 2) *extensibility with unlimited code size* by delegating the handling of some functionalities to other contracts, 3) *code-sharing* by sharing the same logic contract among multiple proxies, etc. We investigate and discuss the use scenarios of proxy design pattern in §IV-B in detail.

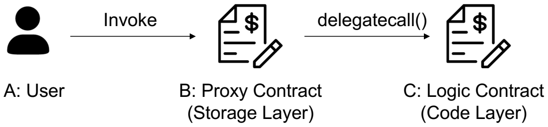


Fig. 1: Workflow of the proxy pattern.

Proxies are prevalent on Ethereum. We find that over 2,031,422 (3.25%) contracts on Ethereum adopt the proxy design pattern. The popular usage of proxies underscores the need to study how the proxy pattern fits in the requirement of DApps, how developers implement proxies, and what kinds of bugs and pitfalls may exist in proxies. *To the best of our knowledge, such a study is still absent.* On the one hand, Bodell *et al.* [8] and Salehi *et al.* [9] studied the upgradeable proxies on Ethereum. However, they confine themselves to upgradeable proxies, while we show that the proxy pattern has several other use scenarios. On the other hand, Ruaro *et al.* [10] proposed techniques to detect a specific type of bug arising in proxies. Our study, in contrast, aims to facilitate an comprehensive understanding of the proxy design pattern, including various use purposes and development pitfalls.

To facilitate our study, we first propose a novel framework, PROXYEX, to detect deployed proxies from bytecode at the Ethereum ecosystem scale. We use PROXYEX to collect a large-scale dataset of 2,031,422 proxies. We manually evaluate PROXYEX on randomly selected 1,000 contracts. Our evaluation shows that PROXYEX can achieve over 99% proxy detection accuracy, 100% precision and over 99% recall. These evaluation results assure the quality of the dataset we collect for the study and consolidate the observations and findings

†These authors contributed equally to this work.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00083

we make. With the large-scale dataset of proxies, we conduct the first systematic study on proxies on Ethereum, aiming to answer the following research questions.

- **RQ1: (Statistics)** *How many proxies are there and How frequently are these proxies used on Ethereum?* We measure the total number and the proportion of distinct proxies on Ethereum, as well as the transactions invoking these proxies, to obtain an overview of the popularity, usage, and traffic of proxies.
- **RQ2: (Purpose)** *What are the major purposes of implementing proxy patterns for DApps?* We manually inspect a sample of proxies and propose a use-purpose taxonomy of proxies. We also propose automated methods to detect each type of proxy on a large scale from our dataset.
- **RQ3: (Bugs and Pitfalls)** *What types of bugs and pitfalls can exist in proxies?* We summarize common pitfalls when implementing proxies and design several pitfall checkers leveraging historical transactions. We use the checkers to detect bugs in all proxies in our dataset.

Findings. The major findings are as follows.

- There are four different use purpose of proxies: namely upgradeability, extensibility, code-sharing, and code-hiding, among which upgradeability proxies and code-sharing proxies compose the majority.
- Over 19.54% proxies are non-upgradeable, i.e., their logic contract, to which the code execution is delegated, cannot be changed. Among the upgradeable proxies, 98.24% of them have never been upgraded in history.
- The proxy contract and logic contracts should use the contract storage consistently with the same semantic interpretation; otherwise, severe bugs may occur.
- Developers often fail to initialize proxy or logic contracts within the same transaction where the contract is deployed, exposing a non-negligible attacking surface for proxies. We have successfully identified a zero-day vulnerability putting over $2M worth of assets at risk.

Contributions. Our study makes the following contributions.

- *Proxy Detection and Dataset:* We propose a novel framework ProxyEX to identify proxies. We collect a large-scale dataset of 2,031,422 real-world proxies on Ethereum as well as their 172,709,392 transactions.
- *Systematic analysis:* We conduct an in-depth and systematic analysis of proxies, propose a taxonomy of proxies and the implementation pitfalls, and make valuable observations and implications on our collected dataset.
- *Bugs and Pitfalls:* We identify three common pitfalls in proxies and conduct a semi-automated detection for such pitfalls. We found a high-value zero-day vulnerability affecting assets over $2M.

Data availability. We released the implementation of ProxyEX and all data in https://github.com/OSUSecLab/ProxyEx.

II. Preliminaries

A. Ethereum Basics

Each smart contract consists of ***code*** and data ***storage***. The code is immutable once deployed. Users invoke functions of the contract code, which may load or modify data in the contract storage. The storage is a low-level data store, comprised of a list of ***storage slots*** indexed with an integer between 0 and 2^{256}. Each storage slot may store 32 bytes of data. Solidity, the most popular programming language of smart contracts, allows developers to declare high-level ***state variables*** of a contract, which internally encodes and stores the value in the low-level storage. Users interact with smart contracts by sending transactions. Each transaction may specify a function (with an 4-byte function signature hash) to be called. If the specified function is not found, a special `fallback` function in the contract will be executed. Functions in one smart contract may call other smart contracts. **`CALL`** and **`DELEGATECALL`** are two different calls between contracts. `CALL` invokes the code of another contract and execute in the context of the callee contract (i.e., modifying the callee's storage). `DELEGATECALL` fetches the code of another contract and execute in the context of the caller contract (i.e., modifying the caller's storage).

B. Proxy Patterns

The proxy pattern (Fig. 1) involves a proxy contract interfacing with users and a logic contract holding the actual implementation code. The proxy contract leverage the `DELEGATECALL` opcode of Ethereum Virtual Machine (EVM) to delegate the code execution to the logic contract, while the data are kept in the storage of the proxy contract. By separating the proxy contract and the logic contract, it enables applications such as upgradeability smart contract, where the logic contract can later be upgraded to a new version, and code-sharing contracts, where multiple proxy contracts are delegating the execution to the same logic contract. We discuss more use scenarios of proxy patterns in §IV-B.

EIP-1967: Proxy Storage Slots [11] is a standard on Ethereum to specify designated storage slots where specific information of proxies should be stored. One example information in proxy is the address of the logic contract. EIP-1967 defines the storage slot for logic contract address as `uint256(keccak256('eip1967.proxy.implementation'))-1)`, which is ensured not allocated to any high-level state variables of the contract, thus preventing storage collsions. We discuss storage collision pitfalls in detail in §IV-C. EIP-1967 also specifies a standard storage slot for other proxy information, such as the admin address. EIP-1967 facilitates proper extraction and display of proxy information by clients like block explorers (e.g., Etherscan [12]) for end users, while logic contracts can choose to utilize it optionally.

III. Data Collection

A. ProxyEX

To facilitate data collection, we design a system, ProxyEX (shown in Fig. 2), to detect proxy contracts from bytecode

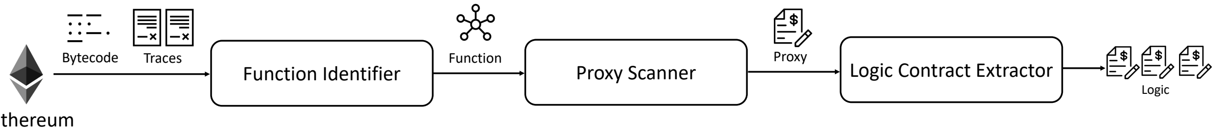
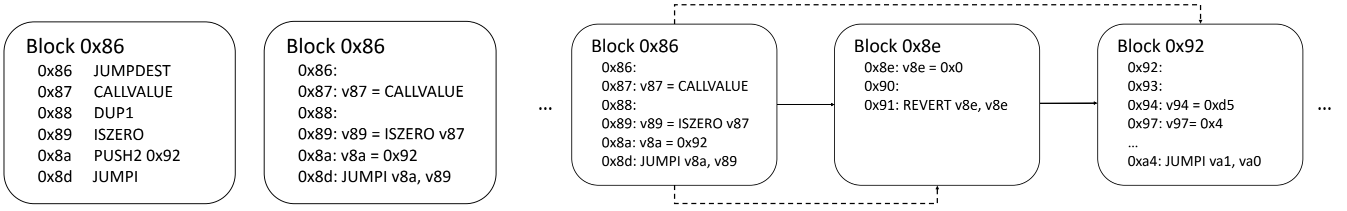


Fig. 2: The workflow of ProxyEX.



```
Block 0x86
  0x86    JUMPDEST
  0x87    CALLVALUE
  0x88    DUP1
  0x89    ISZERO
  0x8a    PUSH2 0x92
  0x8d    JUMPI
```

```
Block 0x86
  0x86:
  0x87: v87 = CALLVALUE
  0x88:
  0x89: v89 = ISZERO v87
  0x8a: v8a = 0x92
  0x8d: JUMPI v8a, v89
```

```
Block 0x86
  0x86:
  0x87: v87 = CALLVALUE
  0x88:
  0x89: v89 = ISZERO v87
  0x8a: v8a = 0x92
  0x8d: JUMPI v8a, v89

Block 0x8e
  0x8e: v8e = 0x0
  0x90:
  0x91: REVERT v8e, v8e

Block 0x92
  0x92:
  0x93:
  0x94: v94 = 0xd5
  0x97: v97= 0x4
  ...
  0xa4: JUMPI va1, va0
```

(a) Opcode sequence. (b) IR. (c) GCFG with IR. Solid line: fall-through edge; dotted line: jump-related edge.

Fig. 3: Illustration of the first three steps of Function Identifier.

and extract logic contract addresses from transactions. It has three major components: 1) Function Identifier, which extracts function-based control flow graphs (CFGs) based on register-based intermediate representation (IR), 2) Proxy Scanner, which detects proxy contracts based on bytecode analysis, 3) Logic Contract Extractor, which utilizes transaction-level analysis to extract Logic contracts.

1) Function Identifier: Given a smart contract bytecode, Function Identifier identifies the function boundaries and extracts function-based control-flow and data-flow graphs as follows. This is necessary for detecting proxy contracts, since we need to 1) identify the `fallback()` function, which is a necessary condition for proxy contracts; and 2) analyze cross-function calls, as the `fallback()` function may call another function to execute `DELEGATECALL`.

Step 1: Constructing basic blocks from `Opcodes`. First, Function Identifier disassembles the bytecode into a sequence of `Opcodes`. After that, Function Identifier divides the `Opcode` sequence into basic blocks using control-flow transfer `Opcodes` as delimiters. There are two types of control-flow transfer `Opcodes`: 1) `JUMP`-related `Opcodes`, including `JUMP`, `JUMPI`, `JUMPDEST`; `JUMP` and `JUMPI` indicate the end of a block, while `JUMPDEST` denotes the beginning of a block. 2) `STOP`-related `Opcodes`, including `STOP`, `RETURN`, `REVERT`, `INVALID` and `SELFDESTRUCT`; they all indicate the end of a block. Note that while call-related `Opcodes` such as `CALL` and `DELEGATECALL` are control-flow transfer `Opcodes`, they are not used as delimiters for creating basic blocks, since there is no callee information available from the bytecode. One example is shown in Fig. 3a.

Step 2: Converting the `Opcode` sequence into register-based intermediate representation (IR). Since EVM `Opcodes` are stack-based operations, extracting control-flow and data-flow information can be challenging. To facilitate further analysis, we build upon Vandal [13], a static program analysis framework for Ethereum smart contract bytecode. Vandal decompiles EVM bytecode into a register-based IR that encodes the program's control flow graph. We chose Vandal due to its robustness and wide adoption by prior works such as TxSpector [14] and MadMax [15]. To be more specific, we simulate the EVM stack operations step-by-step, with producing the global data flow. We generate a new register for every new operand used by an `Opcode`, as well as the operation involving multiple operands (*e.g.*, `JUMPI`). In this way, every stack-based operation will be converted into a register-based operation, as shown in Fig. 3b. For some `Opcodes` (*e.g.*, `JUMPDEST`), they do not generate new values; thus, there is no corresponding register assigned for such `Opcodes` (*e.g.*, empty for `0x86`). For `Opcodes` that generate new values, a register will be given to represent the value (*e.g.*, `v89` for `ISZERO`).

Step 3: Constructing the global control flow graph (GCFG). After the IR is generated, Function Identifier will further construct the GCFG. GCFGs are constructed based on specific opcodes, implemented based on Gigahorse [16]. Gigahorse is a decompiler designed for EVM bytecode, transforming bytecode into a high-level 3-address code representation. For example, when the opcode `JUMP` is encountered, it indicates a change in code execution from one place to another. This allows us to split the execution flow into separate blocks whenever a `JUMP` is encountered. In the GCFGs, the nodes are the basic blocks, and the edges are either fall-through edges or jump-related edges. There are two types of edges: 1) fall-through edges, and 2) jump-related edges. Fall-through edges can be easily added by connecting basic blocks according to the original order. With the IR, we can obtain the targets of jump-related `Opcodes`, which are the values of the `JUMPDEST`. Function Identifier adds a GCFG edge between a basic block ending with `JUMPI` or `JUMP` and the basic block address (i.e., a constant value) held by the variable, which is used to denote the jump target. As shown in Fig. 3c, the solid lines represent fall-through edges, while the dotted lines represent the jump-related edges. For example, block `0x86` jumps to `0x92` if the condition `v89` is satisfied, which means the variable `v87` is `0`; otherwise, block `0x86` will go to the fall-through edge and go to block `0x8e`.

Step 4: Recovering function-based control flow graphs (FCFGs). The final step is to recover FCFGs from the GCFG by identifying function bodies and control/data flows. First, Function Identifier locates function entries by analyzing the EVM function dispatcher. The dispatcher compares each contract function signature with the `input`. If a match is found, control jumps to the function's entry block; otherwise, it executes the `fallback()` function or ends. If Function Identifier finds a function without a signature at the end, it is the `fallback()` function.

2) Proxy Scanner: Proxy Scanner aims to determine whether a given contract is a proxy contract based on the identified functions and FCFGs. To identify the proxy from bytecode, Proxy Scanner leverages the following detection rules based on our observations (in supplementary material).

Rule 1: Locating the `DELEGATECALL` in the `fallback()` function. Proxy Scanner will iterate all the identified functions, to find whether there exists a `fallback()` function; if yes, Proxy Scanner further checks whether there is a `DELEGATECALL` in this specific function.

Rule 2: Ensuring the input for `DELEGATECALL` is derived from the parameters of the `fallback()` function. Before executing `DELEGATECALL`, it is necessary to execute a specific `CALLDATACOPY` operation to fill the memory with the call data, which represents the user input. Therefore, Proxy Scanner checks whether `CALLDATACOPY` is called before `DELEGATECALL` to fill the memory used by `DELEGATECALL`.

Rule 3: Ensuring the returned success status of `DELEGATECALL` is checked and the return data is handled case by case (return when success and revert when fail). After a `DELEGATECALL`, the contract either return (status is 1) or revert (status is 0). Proxy Scanner will fetch the variable of the status returned by the `DELEGATECALL` and check whether the variable is used by a `JUMPI`. If status is 1, the contract will go to a `STOP`/`RETURN`; otherwise, the contract will go to a `REVERT`.

3) Logic Contract Extractor: Since the source code is not available, it is hard to extract the logic contract address directly from the bytecode. We can try to extract Logic contracts from the parameters of `DELEGATECALL` [10], but this method is error-prone since we are not sure which `DELEGATECALL` is really the one that delegates to the Logic contract. To tackle this, we propose to use transaction information to assist our analysis. After Proxy Scanner confirms that a smart contract is a proxy, Logic Contract Extractor will extract all the historical logic contracts used by this proxy via transaction-level analysis.

Logic Contract Extractor enforces rules to identify transactions where the proxy `DELEGATECALL`s the Logic contract from all proxy transactions. We term these transactions as *logic-delegating transactions*. Extracting target addresses of such transactions yields the Logic contracts. We derive the following rules from our observations (in supplementary material) to identify *logic-delegating transactions*.

Rule 1: Checking whether the `fallback()` function is called in a transaction. Logic Contract Extractor first extracts the list of function signatures (8-byte string, e.g., `0x12345678`) from the bytecode. This function list contains all functions that are defined in the proxy contract. After that, Logic Contract Extractor checks whether the function signature of the transaction is in the signature list (the first 8 bytes of the `input`); if not, the transaction will execute the `fallback()` function, based on the execution logic of the EVM (§II).

Rule 2: Checking the parameters from transaction traces. Logic Contract Extractor checks the first two transaction traces to see whether 1) an EOA `CALL`s the proxy contract in the 1st trace; 2) the proxy contract `DELEGATECALL`s another contract in the 2nd trace; 3) the `input` of 1st and 2nd traces are the same.

If a transaction matches the above rules, it is a *logic-delegating transaction*. Note that for the remaining transactions that are not *logic-delegating transactions*, their functionalities are hard to determine. For example, a transaction may be used for performing an upgrade, or updating the owner of the proxy contract. In this paper, we do not implement more rules to distinguish them, since finding *logic-delegating transactions* would be sufficient for extracting Logic contracts.

Extracting Logic contract addresses. After identifying all *logic-delegating transactions*, Logic Contract Extractor extracts all target addresses of `DELEGATECALL`s from every *logic-delegating transaction*, and constructs the set of Logic addresses after deduplication.

B. Correctness of ProxyEX

We developed a prototype of ProxyEX. To evaluate its effectiveness, we randomly selected 1,000 contracts, split into 548 proxies and 452 non-proxies to ensure an unbiased dataset. With a 95% confidence level, the error margin is 4.19% for proxies and 4.61% for non-proxies. Initially, we manually inspected the source code to evaluate ProxyEX 's accuracy. However, 549 contracts lacked source code. In these cases, we inspected the decompiled code using the Online Solidity Decompiler [17], also used by other works [14], [18]. We analyzed ProxyEX 's detection results for 1,000 contracts with a 60-second timeout, comparing them with our ground truth labels. ProxyEX misclassified one proxy as non-proxy due to a timeout caused by nested loops identifying data flow dependencies. Overall, with only one false negative, ProxyEX achieved 100% precision and over 99% recall.

C. Datasets

Data collection involves utilizing Google BigQuery APIs [19], which facilitate querying Ethereum contracts and transactions. For instance, to gather transactions associated with all proxy contracts, we develop SQL code tailored for querying these specific datasets. In total, we have collected the following datasets: **1) Proxy contracts.** We collect all the on-chain smart contract bytecode as of September 10, 2023. In total, we have 62,578,635 smart contracts. We apply ProxyEX on the smart contract bytecode; there are 2,031,422 proxy addresses in total (3.25%). The average detection time of proxy and non-proxy contracts are 14.85 seconds and 3.88 seconds, respectively. **2) Transactions and traces.** We gather all the transaction traces associated with a `DELEGATECALL` sent from the proxy contracts (i.e., *logic-delegating transactions*) as of September 10, 2023. We collect a 3-tuple `{FromAddr, ToAddr, CallType}` for every trace, which we

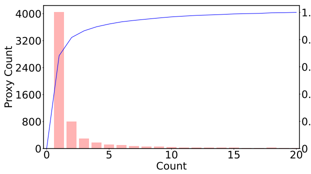


Fig. 4: Bytecode Duplication

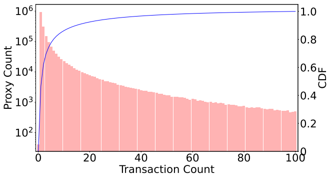


Fig. 5: Transaction Count

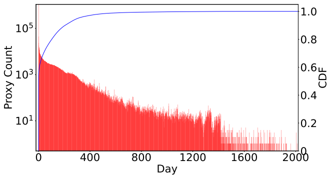


Fig. 6: Lifespan

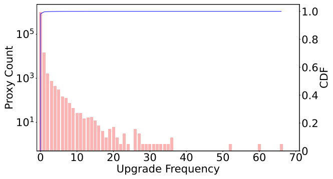


Fig. 7: Upgrade Frequency

subsequently aggregate into transactions. In total, we collect 172,709,392 transactions for all 2,031,422 proxy contracts.

IV. Evaluation Results

In this section, we present our evaluation results based on the datasets to answer three research questions: *RQ1: Statistics*, *RQ2: Purpose*, and *RQ3: Bugs and Pitfalls*.

A. *RQ1: Statistics*

1) Bytecode Duplication: Among 2,031,422 proxy contracts, only 6,254 bytecodes are unique, with 4,039 (64.6%) being exclusive and unduplicated. Thus, just 0.20% of proxy contracts hold unique bytecode. In total, 1,868 (29.86%) bytecodes are shared by 20 or fewer contracts. As shown in Fig. 4, this includes 347 (5.54%) bytecodes shared by over 20 contracts. The top three bytecodes are used by 1,546,495, 292,285, and 19,069 proxy contracts, respectively, and they are standard proxy implementations. For example, the leading contract *OwnableDelegateProxy* integrates ownerable and upgradeable features, requiring the owner's authorization to upgrade its logic contract.

2) Transaction Count: The 2,031,422 proxy contracts are related to 172,709,392 logic-delegating transactions. There are 1,982,378 (97.59%) proxy contracts with less than 100 transactions, and their transaction count is shown in Fig. 5. 1,688,372 (83.11%) of them have less than 10 transactions. There are only 24 proxy contracts having over a million transactions. Among the top 10 addresses, five are token contracts, three are bridge contracts, and two contracts are Decentralized Finance (DeFi) applications. Specifically, the top proxy is the token contract USDC [20] and it has the most transactions (65,497,336), accounting for 37.92% of the total transactions in our dataset.

3) Lifespan: We define the *lifespan* (in days) of a proxy as the duration between its initial `DELEGATECALL` to a logic contract and its final `DELEGATECALL`. If a proxy never `DELEGATECALLs` to a logic contract, its lifespan is considered zero. As shown in Fig. 6, 1,250,750 (61.57%) of the proxies have a lifespan less than 10 days; 4,733 proxies have a lifespan of more than 3 years. The proxy with the longest lifespan is the EventsHistory contract [21] of `Ambisafe Operations`, which has been active since July 18, 2016.

> **Finding 1.** The majority of proxies exhibit identical bytecode (99.80%), involve fewer than 10 associated delegate-calling transactions (83.11%), and maintain a short lifespan of less than 10 days (61.57%).

```
contract upgradeableProxy {
    fallback() external {
        address(storage[0x0a]).delegatecall(msg.data);
    }}
contract Logic {
    address _logic;
    function upgradeTo(address logic) public onlyOwner {
        storage[0x0a] = logic;
    }}
```

Fig. 8: An example of upgradeable proxy.

B. *RQ2: Purpose*

Given the popularity of proxies, it is vital to understand the use purpose of proxies on Ethereum. We conduct a manual inspection on a sample (95% confidence and 5% error margin) of 385 proxies and categorize the purpose of proxies into four types, namely upgradeability, extensibility, code-sharing, and code-hiding. Then, we design automated classifiers to scalably categorize all proxies in our dataset. We discuss each use purpose and present the results below.

1) Upgradeability: Smart contracts on Ethereum are immutable and their code cannot be modified once deployed. This brings many difficulties for developers to maintain a DApp, e.g., upgrading the contract logic to new versions, fixing bugs and vulnerabilities found after deployment, etc. The proxy pattern can be leveraged by developers to alleviate this issue. In upgradeable proxies, the proxy contract stores the data of the DApp, while the code in the proxy contract simply delegates all invocations to a logic contract via `DELEGATECALL`. **No business logic is implemented in the proxy contract and the code in the logic contract executes the business logic on top of the data in the proxy contract.** Fig. 8 shows an example upgradeable proxy. The address of the logic contract is stored in the storage slot `0x0a`, and all invocations to the proxy are delegated to the logic contract in the `fallback` function. A function `upgradeTo` is available to upgrade the logic by assigning a new address to the storage slot `0x0a`. Note that both proxy and logic contracts share and operate on the storage data of the proxy contract.

Classification. The Ethereum community has established several standards for implementing upgradeable proxies, including Transparent Upgradeable Proxy [22] and Universal Upgradeable Proxy Standard (UUPS) [23]. We classify a proxy into to upgradeable proxy if there exists a function that can change the address of the logic contract.

Result. We successfully identified 1,634,396 upgradeable proxies among all proxies in our dataset. Note that not all proxies are upgradeable [9], e.g, minimal proxies [24] and

```
contract TokenProxy {
    mapping(address=>uint) balances;
    function balanceOf(address a) public returns(uint) {
        return balances[a]
    }
    function transfer(address to, uint a) public {
        balances[msg.sender] -= a;
        balances[to] += a;
    }
    fallback() external {
        address(storage[0x0a]).delegatecall(msg.data);
    }}
contract TokenExtension is TokenProxy {
    function burn(address a, uint n) public onlyOwner {
        balance[a] -= n;
    }}
```

Fig. 9: An example proxy to achieve extensibility [28].

DELEGATECALL forwarders [25]. We found that a non-negligible portion (19.54%) of proxies are not upgradeable.

We also measure the upgrade frequency of upgradeable proxies. The upgrade frequency is the number of times a proxy upgrades its logic contract. The upgrade frequency for every proxy is essentially the number of different logic contracts minus one. Note that here we only count the *effective* logic contracts, i.e., the logic contracts that have been used for at least once. If an upgrade introduces a new Logic contract, but it is never used (no *logic-delegating transaction*), it will not be included in our dataset. Fig. 7 presents the upgrade frequency. We find that a majority (over 98.24%) of the upgradeable proxies have never upgraded their logic contracts. 1.75% of them have upgraded at least once but less than 20 times. The proxy with the highest upgrade frequency [26] has 67 unique logic addresses, *i.e.*, has been upgraded 66 times.

Finding 2. Most proxies are upgradeable, while a non-negligible portion (19.54%) are non-upgradeable. However, a majority of upgradeable proxies (98.24%) have never been upgraded after deployment in history.

2) Extensibility: The immutability of smart contracts also limits the extensibility of DApps. The Ethereum community has many application standards for DApps to implement, e.g., ERC-20 (fungible tokens), ERC-721 (NFT), etc. However, these standards only cover basic functionalities and DApps usually need to extend the standards with additional functionalities. We observe that many contract developers leverage proxy patterns to gradually extend the functionalities of their DApps. **The standard basic functionalities are implemented in the proxy contract, while all other non-standard invocations are delegated to the logic contract, which implements extended functionalities.** Fig. 9 shows an ERC-20 [27] contract with extended functionalities via proxy pattern. The standard functionalities (e.g., `transfer` and `balanceOf` in ERC-20) are implemented in the proxy contract, while any invocations calling non-ERC-20-standard functions are forwarded to the logic contract `TokenExtension` (e.g., `burn`) in the `fallback` function of the proxy contract.

Extensibility proxies are designed to balance the trustlessness and extensibility of the DApp. Ideally, the smart contract code should be immutable so that users do not need to trust the contract owners/developers to be honest (and not maliciously modify the logic). Otherwise, contract users may be at risk of rug pull or other centralization risks [29]. By adopting an extensibility proxy pattern, contract developers can keep a balance between trustlessness and extensibility, where the core functionalities implemented in the proxy contracts (e.g., `TokenProxy` in Fig. 9) are immutable while the extended functionalities in logic contracts (e.g., `TokenExtension`) are flexible to update as needed.

Classification. Extensibility proxies may also update the address of the logic contract, which contains the extended functionalities. The key difference is that extensibility proxies implement part of the functionalities in the proxy contract. Therefore, we classify a proxy to be an extensibility proxy if not all business logic invocations are forwarded to the logic contract, i.e., some invocations to the contract are handled by the proxy contract, while some are forwarded to the logic contract. To detect extensibility proxies, our heuristic is that in extensibility proxies, the code in the proxy contract and logic contract must share the data in the proxy. We replay all historical transactions of each proxy and collect extensibility proxies whose proxy and logic contracts have ever accessed the same storage slot in history.

Results. We successfully identified 32 proxies satisfying the aforementioned criteria. We manually investigated each of them and found that 31 of them are true extensibility proxies. The other proxy is not an extensibility proxy but satisfies our classification criteria in that it is an implementation bug. Two different state variables in proxy and logic contracts accidentally use the same storage slot (which they should not) due to an implementation bug. We will further discuss this kind of bug in §IV-C1.

Finding 3. The proxy pattern can be leveraged to achieve the extensibility of functionalities of smart contracts. However, implementing business logic in both proxy and logic contracts is error-prone and requires careful design and correctness audit.

3) Code-sharing: In DApps, it is often the case that multiple smart contracts are deployed with the same code to operate on different data. For example, for decentralized token exchange DApps (e.g., Uniswap), developers need to deploy exchange contracts for each pair of exchangeable tokens. Deploying smart contracts induces non-negligible costs for developers, proportional to the size of the smart contract code with exactly the same logic. Proxy patterns can be leveraged to share code between smart contracts and save deployment costs for developers. **In code-sharing proxies, multiple proxy contracts may delegate their logic execution to the same logic contract.** The code is shared in the logic contract while the data that the shared logic operates are stored in individual proxy contracts. Users can continue to interact with the proxy contracts as if there is no code sharing. Fig. 10

```
contract Proxy1 {
    fallback() external {
        address(storage[0x0a]).delegatecall(msg.data);
    }}
contract Proxy2 {
    fallback() external {
        address(storage[0x0a]).delegatecall(msg.data);
    }}
contract SharedLogic {
    ERC20 token0, token1;
    function swap(amount x) public {
        // swap x amount of token0 to token1
        ...
    }}
```

Fig. 10: An example of code-sharing using the proxy pattern.

shows an example of code-sharing proxy. Both `Proxy1` and `Proxy2` delegate their logic to the `SharedLogic` contract, which provides token exchange functionality. The data of the two proxies are separate, i.e., `Proxy1` and `Proxy2` may have different values for variable `token0` and `token1` to swap different pairs of tokens.

Classification. The major characteristic of code-sharing proxies is that multiple proxies share the same logic contract. To identify code-sharing proxies, we enumerate all proxies in our dataset and check the addresses of their logic contracts. If a logic contract is used by more than one proxy, then those proxies are code-sharing proxies.

Results. We have identified 1,137,317 code-sharing proxies in our dataset. Based on the logic contract they share, the proxies can be separated into different clusters. The size of each cluster is the number of proxies in the cluster; each cluster has a size larger than or equal to 2. There are in total 3,309 code-sharing proxy clusters. 45 (1.35%) clusters have a size larger than 500. There are 705 (21.31%) clusters whose size is larger than 10, and 151 (4.56%) clusters whose size is larger than 100. The most shared logic contract named `AuthenticatedProxy` [30] is shared by 943,601 proxies.

Finding 4. Proxies often share the logic contracts. We identified 1,137,317 code-sharing proxies in our dataset. The top shared logic contract is shared by 943,601 proxies.

4) Code-hiding: Many blockchain explorers, like Etherscan, utilize the EIP standard to implement slots to identify and display the logic contracts. Users and security researchers rely on such explorers, especially Etherscan, to check the source code and labels of on-chain contracts [31]–[33]. To escape from scrutiny by users or other third parties, **malicious actors may hide their malicious logic behind a proxy and deceive blockchain explorers or monitoring tools so that it appears to be a legitimate logic contract** [34]. Such deception is possible in that auditors usually rely on EIP-1967 [11], which defines the specific storage slot where the address of the logic contract is stored, to identify the logic contract of a proxy. Malicious proxies may store an address of a legitimate logic contract in the specific slot defined in EIP-1967, but the actual execution is delegated to another logic contract, which contains malicious logic, at runtime.

```
contract CodeHidingProxy {
    bytes32 eip1967slot = keccak256('eip1967.proxy.implementation') - 1;
    fallback() external {
        storage[eip1967slot] = address(Legitimate);
        address(Malicious).delegatecall(msg.data);
    }}
contract Legitimate {...}
contract Malicious {...}
```

Fig. 11: An example of using the proxy pattern to hide logic.

Note that the developers of code-hiding proxy contracts can be either malicious or benign; it is hard to distinguish in reality due to the anonymity of contract developers. However, regardless of their intentions, their misuse of certain EIP standard implementation slots misleads users. This would cause mis-identification of logic in blockchain explorers and induce security risks such as honeypot attacks [35], as the actual contract owner may deliberately hide malicious logic. Fig. 11 shows an example contract that hides its logic behind a proxy, deceiving blockchain explorers with a fake logic contract address.

Classification. Following the existing practice by Forta Network [36], we identify malicious proxies by checking whether a proxy is delegating its execution to the logic contract, whose address is different from what it claims in the storage slot defined by EIP-1967. We replay all historical transactions of the proxy and compare the address of the actual logic contract being executed to the address stored in that specific storage slot. If they are different, we consider that the proxy is dishonestly hiding its implementation and may be a malicious proxy.

Results. We successfully identified 1,213 proxies that delegate their execution to a logic contract other than the one claimed in the standard storage slot defined by EIP-1967 [11]. Since smart contracts deployed on the blockchain are anonymous, we are not able to confirm the underlying purpose of these proxies hiding logic contracts. Nevertheless, we do find that popular blockchain explorers like Etherscan [12] indicate false logic contracts for these proxies. For instance, for the proxy contract at address `0x9276635ec39c72866f3cf70298efe5501eb5dcdf1`, Etherscan indicates its logic contract as `0xbcb7549e7af77bce0d1bca1a5ef679594e9f2a87`, while the actual executed logic contract is `0x29e45aabc905056162f7521005c6a1919ae6a32c`.

Finding 5. 1,213 contracts hide their logic behind a proxy and deceive blockchain explorers with a fake logic address.

Implication. Users are advised to carefully identify the actual logic contract by simulating the execution before sending transactions to proxy contracts.

C. RQ3: Bugs and Pitfalls

For RQ3, we aim to investigate the common bugs in proxies. We first summarize three kinds of common pitfalls in implementing proxies and then identify such bugs in proxies with semi-automated processes. Our goal is to

```
contract AudiusAdminUpgradeabilityProxy {
    address private proxyAdmin;
    function upgradeTo(address logic) external {
        require(msg.sender == proxyAdmin,
            ERROR_ONLY_ADMIN);
        _upgradeTo(logic);
    }}
contract Governance {
    bool private initialized;
    bool private initializing;
    modifier initializer() {
        require(initializing || !initialized);
        _;
    }
    function initialize(address registry, address
         guardian) initializer public {
        ...
    }}
```

Fig. 12: A simplified version of the vulnerable proxy in Audius [37].

reveal the prevalence of such bugs on Ethereum and offer a comprehensive understanding of them in real-world proxies.

1) Proxy-logic Collision: Many proxies implement their business logic in both proxy contracts and logic contracts (e.g., extensibility proxies in §IV-B2). **If the logic contract uses the same storage slot as the code in the proxy contract but with different semantic interpretations, bugs occur.** The hack of Audius [37] on July 2022 was caused by such storage collision between proxy and logic contracts, inducing over $1.1M of loss. Fig. 12 shows a simplified version of the vulnerable proxy in Audius. The contract `AudiusAdminUpgradeabilityProxy` is the proxy contract, which delegates its business logic to contract `Govercance` (logic contract). The contracts have proxy-logic storage collision between the variable `proxyAdmin` in `AudiusAdminUpgradeabilityProxy` and variables `initialized/initializing` in `Governance`. Note that the values of `initialized` and `initializing` are packed together in one storage slot, each occupying one byte. At runtime, both `proxyAdmin` and `initialized/initializing` point to the storage slot `0x0` of the proxy contract. However, the storage value is interpreted as different types and meanings in the proxy and logic contracts. In the Audius attack, the `Governance` contract has already been initialized by the developers and should not be initialized again. However, the variable `initializing` loads the second byte of storage slot `0x0` as a boolean value, which is `0xab`, the second byte of the `proxyAdmin` address on the blockchain. As a result, the `Governance` contract can always be initialized again and malicious users can exploit this to make profits.

Detection. Similar to the classification of extensibility proxies, we detect proxy-logic collision by replaying all historical transactions of each proxy and checking whether the code in the proxy contract and the code in the logic contract may access the same storage slots in transaction execution. Note that if proxy and logic contracts do not have overlaps in storage slots accessed, they will never collide. We only consider the write access to the storage slots, i.e., write-write conflict, since the read access in either the proxy or logic contract does not influence the execution of other contracts. After the automated detection of write-write conflicts, we manually inspect the filtered proxies to check whether they are vulnerable, i.e., having proxy-logic collisions.

Evaluation. To ensure the correctness of the detection of write-write conflicts, we manually evaluate the correctness of our detector by sampling 100 transactions of proxies and manually check if the write access to storage slots in transactions is correctly captured by our detector or not. We use Phalcon [38], a popular transaction trace explorer, as the ground truth of storage access. The manual check shows 100% accuracy in storage access detection. Note that the write-write conflict detector is only meant to automatically filter proxies that potentially have proxy-logic collisions before we manually inspect the contract and identify bugs.

Results. We identified 32 proxies that contain write-write conflicts on storage between the proxy and logic contract. After the manual check, it turns out that only one proxy is truly vulnerable, while all other 31 proxies are benign ones. Those benign ones are extensibility proxies as we discussed in §IV-B2. The reason behind a high false positive rate is that our detection does not check whether the proxy and logic contracts interpret the shared storage slot in the same way. The write-write conflict is only buggy if the interpretation of storage values is different, like the Audius contract; otherwise, it is benign. We leave the more precise detection of storage collision problems for future work.

Finding 6. The proxy is buggy if the proxy and logic contracts use the same storage slots and the semantics of the usage are different.

Implication. Proxy contract developers should avoid declaring state variables in proxies and only store data in special storage slots as suggested by EIP-1967 [11], which are collision-free, in proxy contract code. For extensibility proxies, the logic contract should always inherit the proxy contract so that the logic contract always uses values in storage slots in the same way as the proxy contract.

2) Logic-logic Collision: **When proxies update the address of their logic contract, the new logic contract may be incompatible with the old version.** In other words, the new logic contract may interpret the storage data in a different way than the previous logic contract, causing unexpected behaviors. The attack on Shata Capital [39] on February 2023 exploited the logic-logic storage collision. Fig. 13 shows the code snippets that have storage collisions between the new and old versions of logic contracts. Shata Capital upgraded the logic contract from `OldEFVault` to `NewEFVault`. The new logic contract `NewEFVault` changes the declaration of state variables. In smart contracts, state variables are assigned storage slots according to their declaration order. As a result, the variable `assetDecimal` in the new logic contract `NewEFVault` reads the storage slot that previously stored the value for the variable `maxDeposit` in the `OldEFVault` contract. After the logic upgrade, the variable `assetDecimal` contains

```solidity
contract OldEFVault {
    ...
    string public constant version = "3.0";
    uint256 public maxDeposit;
    uint256 public maxWithdraw;
    bool public paused;
    ...
}
contract NewEFVault {
    ...
    string public constant version = "4.0";
    uint256 private assetDecimal;
    uint256 public maxWithdraw;
    uint256 public maxDeposit;
    bool public paused;
    ...
}
```

Fig. 13: A simplified version of the old and new logic contracts that have storage collision in Shata Capital [39].

an unexpected large value (which is `maxDeposit` in the old logic), causing a severe loss of $5.1M.

Detection. The key to detecting logic-logic collision is to determine whether the new logic contracts are compatible with the previous version's logic in terms of storage access. One straightforward approach to check the storage compatibility of two logic contracts is to compare the storage layout of two logics. However, many of the proxies in our dataset do not have source code available thus we cannot obtain their storage layouts. Therefore, we propose to simulate historical transactions on newer logics and check if the storage access patterns of historical transactions remain the same. If a transaction simulated on a newer logic no longer accesses the storage slots that were accessed on the original logic contract on which this transaction was executed in history, it is highly likely that the new logic contract has incompatible storage layouts than the previous logic. To further increase the precision of our detection, we also infer the type of storage data in logic contracts by inspecting the values being written to the storage. If one storage slot is inferred to be different types in two logic contracts, there is highly likely to be a storage collision.

Specifically, given a transaction T originally executed on the logic contract L_n of proxy P in history, we simulate T on proxy P with a newer logic contract L_m, where m, n are version numbers and $m > n$. We record the storage access of T on logic L_n (in blockchain history) and L_m (in our simulation), separately. Then, we check if there exist a storage access that were performed on L_n (original logic) but not on L_m (newer logic). If so, the newer logic is likely to be incompatible with the original logic.

To infer the type of each storage slot in logic contracts, we by default consider each shared storage slot with a union type `bool`|`int`|`address`|`bytes32`, and then gradually narrow down its type by inspecting the values written to this storage slot during transaction execution. Specifically, we assume a common value range of different types as shown in Table I, and discard a type if a value out of the range is written to the storage slot. For instance, if we find that there exists a transaction writting value 100 to storage slot `0x0`, we will infer that this storage slot is not of type `bool`. Note that our assumed value range of types are not the theoretical value range of the corresponding Solidity type. Our assumed value range is aimed to capture the range of common value at runtime and approximate the high-level types in smart contracts. We infer types of each storage slot for different logic contracts separately. If two logic contracts of the same proxy differs in some types of storage slots, there is likely to be a storage collision.

`bool`	`int`	`address`	`bytes32`
$[0, 1]$	$[0, 2^{32})$	$[2^{32}, 2^{160}]$	$(2^{160}, 2^{256})$

TABLE I: Assumed value range of inferred types for storage slots.

Evaluation. Note that the aforementioned transaction replay and type inference are meant to automatically filter out most logic contracts that are not vulnerable before we manually check the logic contracts to identify collision bugs. We eventually use manual analysis to inspect the implementation details of two logic contracts of the same proxy and determine whether they interpret the same storage slot in different semantics, i.e., there are logic-logic collision bugs. As such, it is necessary to evaluate and ensure that the filtering process does not exclude many truly problematic proxies.

We sample 100 pairs of old and new logic contracts excluded by transaction replay and type inference, and check for storage collisions using evm.storage [40]. We find only two pairs with conflicting storage layouts, indicating 98% precision in our automated filtering and a low probability of missing collisions.

Results. We first simulated all transactions proxies on new versions of logic contracts and found that there are 15,176 proxies whose newer logic has different storage access pattern than then old logic. We then infer storage slot types for each logic contract of these 15,176 proxies and identified 588 proxies whose old and new logic have different storage slot types. We further manually checked each of these 588 proxies and confirmed that 15 of them contain true logic-logic collision bugs.

Finding 7. Proxies may suffer from logic-logic storage collision if the newly upgraded logic contract is incompatible with the old version. We identified 15 proxies that have logic-logic collision bugs in their upgrades.

Implication. Developers should keep the storage layout unchanged when upgrading logic contracts. Developers may consider to always inherit the old version of logic contract when developing a new version to ensure the storage compatibility.

3) Uninitialized Proxy: Smart contracts usually initialize their state in the contract constructor (e.g., set the `owner` of the contract), which is executed when the contract is deployed. However, proxies cannot initialize their state using the constructor since the data and code are separated into two contracts (proxy contract and logic contract), i.e., the constructor of the logic contract cannot initialize the data in the proxy contract. To mitigate this issue, proxies usually implement an `initialize` function instead, which needs to

```
contract Proxy {
    function fallback() external {
        address(Logic).delegatecall(msg.data);
    }}
contract Logic {
    address owner;
    bool private initialized;
    function initialize(address _owner) public {
        require(!initialized);
        initialized = true;
        owner = _owner;
    }}
```

Fig. 14: An example proxy with `initialize` function.

be called explicitly after deployment. Fig. 14 shows an example proxy with the `initialize` function. The `initialize` function brings a new attack surface of front-running. **If the `initialize` function is not called within the same transaction as the contract deployment, attackers may front-run the invocation of `initialize` function and initialize the contract for their own use** , e.g., claim the ownership as in Fig. 14. Uninitialized proxies have been exploited many times in history, including the hacks on Parity [41], Aave [42], Teller [43], KeeperDAO [44], Rivermen NFT [44], Harvest Finance [45], and Wormhole [46].

Detection. We detect the uninitialized proxies by trying to invoke `initialize` function right after the deployment transaction of the proxy in the history. The function `initialize` can only be executed once, so if the developer had already initialized the proxy in the deployment transaction, our invocation would fail; otherwise, it indicates that this is an uninitialized proxy subject to front-running attacks. One challenge is that there is no fixed function signature for `initialize` function. The `initialize` functions in different proxies may have different names and parameters. Without knowing the function signature, we cannot craft valid call data to initialize the proxy. To tackle this challenge, we mine all possible `initialize` function signatures on all proxies in our dataset. We assume that for most proxies, the first invocation after the deployment should be calling the `initialize` function. We inspect the transaction history of each proxy and collect a set of 255 distinct invocations of `initialize` functions. In the end, we replay each initializing invocation on each proxy at the moment right after their deployment transaction and check if it is possible to initialize the proxy with an arbitrary user.

Evaluation. We conduct a manual evaluation of the `initialize` functions we inferred automatically to ensure that we do not miss many uninitialized proxies. Specifically, we sample 100 source-available logic contracts from our proxy dataset, manually identify the `initialize` function used by these proxies, and check if they are included in our automatically extracted set of `initialize` functions. Results show that our automated technique identifies 62% of the `initialize` functions. Note that many missed `initialize` functions are specific only in one logic contract and that contract is often never used after deployment. Nevertheless, by replaying each possible initializing invocation, we faithfully identify majority of different forms of `initialize` functions.

Results. We successfully identified 183 proxies that were not initialized in the same transaction as the deployment. Note that although these proxies were subject to front-running attacks by the time of their deployment, they are not necessarily exploitable on the latest state of Ethereum since the proxy may have already been initialized by the developer at present. Hence, we further investigate each of these proxies manually to check whether they are still exploitable at the latest state of Ethereum (January 2024). We found that 103 out of 183 proxies (56.28%) are still exploitable, meaning that anyone can re-initialize these proxies at present. Among 103 exploitable proxies, 81 proxies have never been initialized in history. The rest 22 proxies, although they have already been initialized by their developers, can still be re-initialized due to improper access control to the `initialize` function. Noteworthy, we were able to identify a zero-day vulnerability that is still exploitable and can induce severe consequences. The affected Total Value Locked (TVL) is over $2M. We have reported to the developers who have started working on rescuing the affected assets at the time of writing.

Finding 8. A proxy may be subject to front-running attacks if it is not initialized within the same transaction as its deployment. We identified a zero-day vulnerability confirmed by the developer, which can cause over $2M loss.

Implication. Developers should be aware of the front-running attack risk and initialize their proxies atomically in the deployment transaction, and should not allow afterward re-initialization under any circumstances.

V. Discussion

A. Threats to Validity

We acknowledge several threats to the validity of the results and findings in our study. First, **quality of the proxy dataset**. We collect a large-scale dataset of proxies using ProxyEX. Contracts that are not proxies may be included and some true proxies may be missed by ProxyEX. To mitigate this threat, we conducted an evaluation in §III-A2 to show that ProxyEX achieve a high precision (100%) and recall (99%). Second, **manual inspection**. We categorized the use purposes of proxies (§IV-B) manually, which may induce incompleteness and bias. To mitigate this threat, two of the authors first individually inspected the sampled proxies and proposed their categories independently. Then, the two authors, together with an additional author, discussed and merged their proxy categories and finalize the taxonomy of use purposes with a consensus. Third, **validity of detected bugs**. We designed three detectors in §IV-C and identified several bugs from our dataset. The detected bugs may be false positives. To mitigate this threat, we invite an experienced Solidity programmer with over 4 years of experience to confirm our findings.

B. Usefulness

This study aims to provide valuable insights for researchers, developers and users. **For researchers,** our study guides future research by highlighting the need to focus on various types of proxies, not just upgradeable ones (§IV-B). Different uses of proxies present new challenges in contract design, bug detection, and on-chain security. Our findings can inspire research on design trade-offs of trustlessness and flexibility features in DApps (upgradeability and extensibility proxies, §IV-B1 and §IV-B2) and serve as heuristics for contract testing and bug detection (§IV-C). **For blockchain users**, we offer a comprehensive study to understand the usage and security risks in contract proxies (§IV-B and §IV-C), such as malicious code-hiding proxies (§IV-B4). **For developers,** our study offers practical guidance by summarizing proxy use purposes and helping developers choose the right design patterns (§IV-B). It alerts developers to potential security issues (§IV-C). Developers are advised to properly initialize contract state at deployment (§IV-C3) and carefully avoid storage collisions between both proxy-logic and logic-logic contracts (§IV-C1 and §IV-C2) to improve DApp quality.

VI. Related Work

Prior Works on Proxy Contracts. To the best of our knowledge, prior papers related to proxy contracts mainly focus on upgradeable proxies [8], [9]. Bodell *et al.* proposes UscHunt [8], which is a static analysis framework for detecting upgradeable smart contracts based on smart contract source code. UscHunt performs upgradeable proxy detection on 8 Ethereum-based blockchains; For Ethereum, it detects 5,384 upgradeable proxies from about 500k smart contract source code. Salehi *et al.* [9] defined six smart contract upgrade patterns, and built a measurement framework to study smart contracts in those categories within a one-year transaction dataset. They replayed every transaction and utilized EVM transaction traces for detecting transactions using `DELEGATECALL`, and extracted the bytecode of the *to* address of such transactions as proxy contracts. Recently, Ruaro *et al.* proposed Crush [10], a tool focusing on detecting storage collisions in proxy smart contracts. Similar to [9], Crush solely detects `DELEGATECALL` inside transactions to identify proxy and logic contracts, which is overly general according to the definition of proxy from OpenZeppelin [47].

We compare our work with existing works in Table II. In summary, our work has a broader scope, with three key differences. *1) Comprehensive Coverage.* We cover *all proxy contracts* (upgradeable and non-upgradeable) following OpenZeppelin's official proxy definition [47]; existing works [8], [9] only cover upgradeable ones. As shown in §IV-B, around 19.54% of proxies are non-upgradeable. *2) Large-Scale Study.* We conduct the first large-scale and systematic study on *all proxy contracts*, extracting insights from three aspects: Statistics, Purpose, and Bugs and Pitfalls. This provides a complete view of the proxy landscape, unlike existing works that focus on a single security issue (storage collision in [10]). *3) Methodology.* Our proxy detection methodology significantly differs from existing works. UscHunt [8] detects proxies from source code, but their method is limited since about two-thirds of Ethereum contracts lack source code [48]. Salehi *et al.* [9] and Crush [10] detect proxies based solely on the `DELEGATECALL` action, which is error-prone and leads to many false positives. For instance, the ERC-6357 contract [49] is incorrectly classified as a proxy. We introduce the first approach to precisely identify proxies from contract bytecode.

	Require Source?	Matching OZ's Proxy Def?	Support Non-USCs?	Dataset Duration
UscHunt [8]	Yes	No	No	Mar. 2016 - Jan. 2022
Salehi *et al.* [9]	No	No	No	Sep. 2020 - Jul. 2021
Crush [10]	No	No	Yes	Jul. 2015 - Apr. 2023
This Paper	No	Yes	Yes	Jul. 2015 - Sep. 2023

TABLE II: Comparison of our paper and related works; OZ's Proxy Def: the official proxy definition by OpenZeppelin [47]; Non-USC: non-upgradable smart contracts.

Empirical Analysis on Ethereum. Many researches perform empirical analysis on different aspects of Ethereum, such as the transactions and interactions among different entities [50]–[57]. Miner Extractable Value (MEV) or Block Extractable Value (BEV) have also attracted much interest [33], [58]–[66]. However, none of these works focuses on proxy contracts.

Bug Detection on Smart Contracts. To secure smart contracts before they are deployed, many works study the vulnerabilities of smart contracts and attacks transactions on Ethereum. Various works use static analysis approaches [13], [14], [67]–[80], such as symbolic execution, to uncover vulnerabilities and bugs in smart contracts. Some other works [81]–[91] make use of fuzzing techniques to discover bugs in smart contracts. Due to the severity of smart contract vulnerabilities and DeFi attacks, many researchers tried to present systematization of knowledge papers (SoKs) [33], [92]–[99]. Compared to these works, our work focuses on proxy contracts.

VII. Conclusion

The proxy pattern is an important design pattern in Ethereum smart contracts. In this paper, we present the first systematic empirical study on proxy contracts at the Ethereum ecosystem scale. We build the first framework for detecting proxy contracts from bytecode, and collect a dataset of 2,031,422 proxies as well as their transactions to perform empirical analysis. We first study the basic statistics, and find that code duplication is prevalent in proxies. To study the purpose of proxies, we propose a taxonomy to categorize proxies into four categories – upgradeability, extensibility, code-sharing, and code-hiding, and perform analysis on each type. We further summarize three types of common bugs in proxies and design checkers to detect them in our dataset. This paper provides valuable insights into the current landscape of proxies, which can facilitate future research on different aspects of proxies.

Acknowledgement

This work was supported in part by an Ethereum Foundation Academic Grant, and a generous gift from the Forta Foundation. The Purdue authors were partially supported by an Ethereum Foundation small grant (FY24-1601).

References

[1] V. Buterin, "Ethereum white paper: A next generation smart contract & decentralized application platform," 2013.

[2] Ethereum, "Ethereum.org." https://ethereum.org/en/, 2024.

[3] DappRadar, "Dappradar - ethereum." https://dappradar.com/chain/ethereum?range-cs=all, 2024.

[4] V. Buterin, "Eip-170: Contract code size limit," *Ethereum Improvement Proposals, no. 170*, 2016. https://eips.ethereum.org/EIPS/eip-170.

[5] N. Mudge, "Eip-2535: Diamonds, multi-facet proxy," *Ethereum Improvement Proposals, no. 2535*, 2020. https://eips.ethereum.org/EIPS/eip-2535.

[6] Y. Kambayashi and H. Ledgard, "The separation principle: A programming paradigm," *IEEE Software*, vol. 21, no. 2, pp. 78–87, 2004.

[7] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, 2014.

[8] W. E. Bodell III, S. Meisami, and Y. Duan, "Proxy hunting: Understanding and characterizing proxy-based upgradeable smart contracts in blockchains," in *32nd USENIX Security Symposium*, 2023.

[9] M. Salehi, J. Clark, and M. Mannan, "Not so immutable: Upgradeability of smart contracts on ethereum," *arXiv preprint arXiv:2206.00716*, 2022.

[10] N. Ruaro, F. Gritti, R. McLaughlin, I. Grishchenko, C. Kruegel, and G. Vigna, "Not your type! detecting storage collision vulnerabilities in ethereum smart contracts,"

[11] E. I. Proposals, "Erc-1967: Proxy storage slots." https://eips.ethereum.org/EIPS/eip-1967, 2019.

[12] Etherscan, "Etherscan." https://etherscan.io/, 2024.

[13] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[14] M. Zhang, X. Zhang, Y. Zhang, and Z. Lin, "TXSPECTOR: Uncovering attacks in ethereum from transactions," in *29th USENIX Security Symposium*, 2020.

[15] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[16] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: Thorough, declarative decompilation of smart contracts," in *International Conference on Software Engineering (ICSE)*, 2019.

[17] EtherVM, "Online solidity decompiler." https://ethervm.io/decompile/, 2024.

[18] N. Grech, S. Lagouvardos, I. Tsatiris, and Y. Smaragdakis, "Elipmoc: Advanced decompilation of ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 1–27, 2022.

[19] G. Bigquery, "Ethereum in bigquery: a public dataset for smart contract analytics." https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics, 2018.

[20] Etherscan, "Address." https://etherscan.io/address/0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48, 2023.

[21] Etherscan, "Eventshistory contract address." https://etherscan.io/address/0x60bf91ac87fee5a78c28f7b67701fbcfa79c18ec, 2024.

[22] OpenZeppelin, "The transparent proxy pattern." https://blog.openzeppelin.com/the-transparent-proxy-pattern/, 2018.

[23] G. Barros and P. Gallagher, "Eip-1822: Universal upgradeable proxy standard (uups)," *Ethereum Improvement Proposals, no. 1822*, 2019. https://eips.ethereum.org/EIPS/eip-1822.

[24] P. Murray, N. Welch, and J. Messerman, "Erc-1167: Minimal proxy contract." https://eips.ethereum.org/EIPS/eip-1167, 2018.

[25] V. Buterin, "Delegatecall forwarders: how to save 50-98 contracts with the same code." https://www.reddit.com/r/ethereum/comments/6c1jui/delegatecall_forwarders_how_to_save_5098_on/, 2017.

[26] Etherscan, "Proxy contract address with the highest upgreade frequency." https://etherscan.io/address/0x3d71d79c224998e608d03c5ec9b405e7a38505f0, 2024.

[27] Ethereum, "Erc20 token." https://ethereum.org/en/developers/docs/standards/tokens/erc-20/, 2024.

[28] Etherscan, "Address." https://etherscan.io/address/0x630d98424efe0ea27fb1b3ab7741907dffeaad78, 2024.

[29] F. Cernera, M. L. Morgia, A. Mei, and F. Sassi, "Token spammers, rug pulls, and sniper bots: An analysis of the ecosystem of tokens in ethereum and in the binance smart chain (BNB)," in *32nd USENIX Security Symposium (USENIX Security 23)*, (Anaheim, CA), pp. 3349–3366, USENIX Association, Aug. 2023.

[30] Etherscan, "Authenticatedproxy contract address." https://etherscan.io/address/0xf9e266af4bca5890e2781812cc6a6e89495a79f2, 2018.

[31] L. Chen, J. Peng, Y. Liu, J. Li, F. Xie, and Z. Zheng, "Phishing scams detection in ethereum transaction network," *ACM Transactions on Internet Technology (TOIT)*, vol. 21, no. 1, pp. 1–16, 2020.

[32] S. Li, G. Gou, C. Liu, C. Hou, Z. Li, and G. Xiong, "Ttagn: Temporal transaction aggregation graph network for ethereum phishing scams detection," in *Proceedings of the ACM Web Conference 2022*, pp. 661–669, 2022.

[33] S. Yang, F. Zhang, K. Huang, X. Chen, Y. Yang, and F. Zhu, "Sok: Mev countermeasures: Theory and practice," *arXiv preprint arXiv:2212.05111*, 2022.

[34] F. Community, "Evasion techniques: Report on the continuous monitoring." https://github.com/apehex/web3-evasion-techniques/blob/main/report/forta.pdf, 2023.

[35] L. Academy, "Honeypot crypto scam meaning.." https://www.ledger.com/academy/glossary/honeypot-crypto-scam, 2024.

[36] F. Network, "Evasion bounty: Fake standards." https://forta.notion.site/Evasion-Bounty-Fake-Standards-673a496a7684498a80ca4d07060fb160, 2023.

[37] Audius, "Audius governance takeover post-mortem 7/23/22." https://blog.audius.co/article/audius-governance-takeover-post-mortem-7-23-22.

[38] BlockSec, "Phalcon explorer." https://blocksec.com/explorer, 2024.

[39] Peckshield, "It seemed shata capital's efvault suffered from an upgrade glitch." https://twitter.com/peckshield/status/1630490333716029440.

[40] EVM.storage, "Blockchain search + exploration.." https://explorer.sim.io, 2024.

[41] OpenZeppelin, "The parity wallet hack explained." https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/.

[42] TrailofBits, "Breaking aave upgradeability." https://blog.trailofbits.com/2020/12/16/breaking-aave-upgradeability/.

[43] Immunefi, "Teller bugfix review and bug bounty launch." https://medium.com/immunefi/teller-bug-fix-postmorten-and-bug-bounty-launch-b3f67a65c5ac.

[44] Iosiro, "Perma-brick uups proxies with this one trick (devs hate this!)." https://iosiro.com/blog/openzeppelin-uups-proxy-vulnerability-disclosure.

[45] Immunefi, "Harvest finance uninitialized proxies bugfix review— $200k bounty." https://medium.com/immunefi/harvest-finance-uninitialized-proxies-bug-fix-postmortem-ea5c0f7af96b.

[46] Immunefi, "Wormhole uninitialized proxy bugfix review." https://medium.com/immunefi/wormhole-uninitialized-proxy-bugfix-review-90250c41a43a.

[47] OpenZeppelin, "Openzeppelin docs - proxies." https://docs.openzeppelin.com/contracts/3.x/api/proxy, 2024.

[48] T. Abdelaziz and A. Hobor, "Smart learning to find dumb contracts," in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 1775–1792, 2023.

[49] E. I. Proposals, "Erc-6357: Single-contract multi-delegatecall.." https://eips.ethereum.org/EIPS/eip-6357, 2023.

[50] X. T. Lee, A. Khan, S. Sen Gupta, Y. H. Ong, and X. Liu, "Measurements, analyses, and insights on the entire ethereum blockchain network," in *Proceedings of The Web Conference 2020*, 2020.

[51] T. Chen, Z. Li, Y. Zhu, J. Chen, X. Luo, J. C.-S. Lui, X. Lin, and X. Zhang, "Understanding ethereum via graph analysis," *ACM Transactions on Internet Technology (TOIT)*, 2020.

[52] L. Zhao, S. Sengupta, A. Khan, and R. Luo, "Temporal analysis of the entire ethereum blockchain network," *Proceedings of the Web Conference 2021*, 2021.

[53] Q. Bai, C. Zhang, Y. Xu, X. Chen, and X. Wang, "Evolution of ethereum: A temporal graph perspective," 2020.

[54] A. Said, M. U. Janjua, S.-U. Hassan, Z. Muzammal, T. Saleem, T. Thaipisutikul, S. Tuarob, and R. Nawaz, "Detailed analysis of ethereum network on transaction behavior, community structure and link prediction," *PeerJ Computer Science*, 2021.

[55] D. Lin, J. Wu, Q. Yuan, and Z. Zheng, "Modeling and understanding ethereum transaction records via a complex network approach," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2020.

[56] J. Zanelatto Gavião Mascarenhas, A. Ziviani, K. Wehmuth, and A. B. Vieira, “On the transaction dynamics of the ethereum-based cryptocurrency,” *Journal of Complex Networks*, 2020.

[57] L. Liu, L. Wei, W. Zhang, M. Wen, Y. Liu, and S.-C. Cheung, “Characterizing transaction-reverting statements in ethereum smart contracts,” in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ASE ’21, p. 630–641, IEEE Press, 2022.

[58] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, “Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges,” *arXiv preprint arXiv:1904.05234*, 2019.

[59] K. Qin, L. Zhou, and A. Gervais, “Quantifying blockchain extractable value: How dark is the forest?,” *arXiv preprint arXiv:2101.05511*, 2021.

[60] C. F. Torres, R. Camino, and R. State, “Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the ethereum blockchain,” *arXiv preprint arXiv:2102.03347*, 2021.

[61] Y. Wang, Y. Chen, H. Wu, L. Zhou, S. Deng, and R. Wattenhofer, “Cyclic arbitrage in decentralized exchanges,” *Available at SSRN 3834535*, 2022.

[62] A. Capponi, R. Jia, and Y. Wang, “The evolution of blockchain: from lit to dark,” *arXiv preprint arXiv:2202.05779*, 2022.

[63] J. Piet, J. Fairoze, and N. Weaver, “Extracting godl [sic] from the salt mines: Ethereum miners extracting value,” *arXiv preprint arXiv:2203.15930*, 2022.

[64] B. Weintraub, C. F. Torres, C. Nita-Rotaru, and R. State, “A flash (bot) in the pan: Measuring maximal extractable value in private pools,” *arXiv preprint arXiv:2206.04185*, 2022.

[65] X. Lyu, M. Zhang, X. Zhang, J. Niu, Y. Zhang, and Z. Lin, “An empirical study on ethereum private transactions and the security implications,” *arXiv preprint arXiv:2208.02858*, 2022.

[66] W. Zhang, L. Wei, S.-C. Cheung, Y. Liu, S. Li, L. Liu, and M. R. Lyu, “Combatting front-running in smart contracts: Attack mining, benchmark construction and vulnerability detector evaluation,” *IEEE Transactions on Software Engineering*, vol. 49, no. 6, pp. 3630–3646, 2023.

[67] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, “Online detection of effectively callback free objects with applications to smart contracts,” *Proceedings of the ACM on Programming Languages*, 2017.

[68] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, ACM, 2016.

[69] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, “Finding the greedy, prodigal, and suicidal contracts at scale,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, ACM, 2018.

[70] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, “Securify: Practical security analysis of smart contracts,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2018.

[71] J. Krupp and C. Rossow, “teether: Gnawing at ethereum to automatically exploit smart contracts,” in *27th USENIX Security Symposium*, 2018.

[72] ConsenSys, “Mythril classic.” https://github.com/ConsenSys/mythril-classic, 2022.

[73] TrailOfBits, “Manticore: Symbolic execution tool.” https://github.com/trailofbits/manticore, 2022.

[74] J. Feist, G. Grieco, and A. Groce, “Slither: a static analysis framework for smart contracts,” in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, IEEE, 2019.

[75] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, “Zeus: Analyzing safety of smart contracts,” in *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, 2018.

[76] C. F. Torres, J. Schütte, and R. State, “Osiris: Hunting for integer bugs in ethereum smart contracts,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.

[77] J. Frank, C. Aschermann, and T. Holz, “{ETHBMC}: A bounded model checker for smart contracts,” in *29th USENIX Security Symposium*, 2020.

[78] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, “Smartcheck: Static analysis of ethereum smart contracts,” in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, IEEE, 2018.

[79] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, “Sailfish: Vetting smart contract state-inconsistency bugs in seconds,” in *2022 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2022.

[80] W. Zhang, Z. Zhang, Q. Shi, L. Liu, L. Wei, Y. Liu, X. Zhang, and S.-C. Cheung, “Nyx: Detecting exploitable front-running vulnerabilities in smart contracts,” in *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 146–146, IEEE Computer Society, 2024.

[81] B. Jiang, Y. Liu, and W. Chan, “Contractfuzzer: Fuzzing smart contracts for vulnerability detection,” in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2018.

[82] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, “Echidna: effective, usable, and fast fuzzing for smart contracts,” in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[83] V. Wüstholz and M. Christakis, “Harvey: A greybox fuzzer for smart contracts,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[84] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, “sfuzz: An efficient adaptive fuzzer for solidity smart contracts,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020.

[85] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, “Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2021.

[86] A. Groce and G. Grieco, “echidna-parade: a tool for diverse multicore smart contract fuzzing,” in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[87] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, “Learning to fuzz from symbolic execution with application to smart contracts,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[88] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, “Reguard: finding reentrancy bugs in smart contracts,” in *IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, IEEE, 2018.

[89] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, “Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts,” in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2021.

[90] Y. Fu, M. Ren, F. Ma, H. Shi, X. Yang, Y. Jiang, H. Li, and X. Shi, “Evmfuzzer: detect evm vulnerabilities via fuzz testing,” in *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019.

[91] M. Rodler, D. Paaßen, W. Li, L. Bernhard, T. Holz, G. Karame, and L. Davi, “Ef/cf: High performance smart contract fuzzing for exploit generation,” *arXiv preprint arXiv:2304.06341*, 2023.

[92] N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on ethereum smart contracts (sok),” in *International Conference on Principles of Security and Trust*, Springer, 2017.

[93] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais, “Sok: Decentralized finance (defi) incidents,” *arXiv preprint arXiv:2208.13035*, 2022.

[94] S. M. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. J. Knottenbelt, “Sok: Decentralized finance (defi),” *arXiv preprint arXiv:2101.08778*, 2021.

[95] L. Heimbach and R. Wattenhofer, “Sok: Preventing transaction reordering manipulations in decentralized finance,” in *4th ACM Conference on Advances in Financial Technologies (AFT)*, 2022.

[96] S. Eskandari, S. Moosavi, and J. Clark, “Sok: Transparent dishonesty: front-running attacks on blockchain,” in *Financial Cryptography and Data Security: FC 2019 International Workshops*, Springer, 2020.

[97] N. Ivanov, C. Li, Q. Yan, Z. Sun, Z. Cao, and X. Luo, “Security threat mitigation for smart contracts: A comprehensive survey,” *ACM Computing Surveys*, 2023.

[98] M. Zhang, X. Zhang, J. Barbee, Y. Zhang, and Z. Lin, “Sok: Security of cross-chain bridges: Attack surfaces, defenses, and open problems,” *arXiv preprint arXiv:2312.12573*, 2023.

[99] Z. Zhang, B. Zhang, W. Xu, and Z. Lin, “Demystifying exploitable bugs in smart contracts,” in *Proceedings of the 45th International Conference on Software Engineering*, 2023.