PROXiFY: A Bytecode Analysis Tool for Detecting and Classifying Proxy Contracts in Ethereum Smart Contracts

Ilham Qasse*, Mohammad Hamdaqa*† Björn Þór Jónsson*,
*Department of Computer Science, Reykjavik University, Reykjavik, Iceland
†Department of Computer and Software Engineering, Polytechnique Montreal, Montreal, Canada
*{ilham20,bjorn,mhamdaqa}@ru.is, †mhamdaqa@polymtl.ca

***Abstract*—As Ethereum smart contracts grow in complexity, upgrades are necessary but challenging due to their immutable nature. Proxy contracts enable upgrades without changing contract state, but current detection approaches often rely on source code or transaction history and fail to detect inactive proxies. Detecting these proxies is critical because dormant upgrade paths can be reactivated, introducing risks and potential attacks. We introduce PROXiFY, a lightweight bytecode-based tool that detects and classifies proxy contracts, including inactive ones, without requiring Ethereum nodes, source code, or customized EVMs. PROXiFY achieves a precision of 98.6% and recall of 97.1% on a high-confidence benchmark dataset. A demonstration of PROXiFY can be viewed at https://youtu.be/FuYs22_vosk.**

***Index Terms*—Proxy Contracts, Smart Contracts, Bytecode Analysis, Immutability, Software Maintenance**

I. Introduction

Smart contracts are designed to be immutable, ensuring trust, transparency, and security [1], [2]. Once deployed, these contracts are unchangeable, serving as binding agreements between parties. While this immutability is foundational to implement trust, it poses challenges for decentralized applications (DApps) that require updates and maintenance [3], [4]. As DApps grow in complexity, addressing vulnerabilities and adding new features becomes essential [4], [5]. However, Ethereum lacks a native mechanism to modify deployed contracts, making it difficult to adapt them over time [3], [6].

To overcome this limitation, the smart contract community developed upgradeability mechanisms that allow smart contracts to evolve without losing their state [6]–[8]. The most prominent solution is the proxy pattern, which initially relied on forward proxies to direct calls to a fixed implementation contract, later evolving to support upgradeable proxies. In this model, the proxy contract manages the state, while the implementation contract contains the logic, which can be updated by pointing to a new implementation when necessary [8]–[10]. This allows for seamless contract upgrades without altering the contract address or disrupting user interactions. Standards like EIP-1967,[1] and EIP-1822,[2] have become widely adopted across Ethereum for this purpose [7], [9].

Despite their advantages, proxy patterns introduce risks. Upgradeable proxies can obscure the true behavior of smart contracts, especially if users are unaware of the upgradeability [4], [7]. This lack of transparency may raise security concerns, as frequent updates complicate auditing and leave room for potential vulnerabilities [4]. There is a growing demand for tools to reliably detect, classify, and trace upgradeable proxies to maintain transparency and security within the Ethereum ecosystem [8]–[10].

Current approaches to proxy detection, such as source code or transaction-based analysis, have limitations [7], [11]. Source code analysis is only possible when the code is verified, which applies to a minority of smart contracts on Ethereum [7]. Transaction-based analysis may miss inactive proxies that have not yet been used or upgraded [7], [11]. This is particularly concerning because inactive upgradeable proxies can later be reactivated, exposing users to hidden risks. For example, Audius lost $6 million in July 2022 when a governance contract with a lingering upgrade path was exploited to drain funds.[3] Detecting such latent upgrade mechanisms before they are abused is therefore critical for auditors and protocol maintainers.

This paper introduces **PROXiFY**,[4] a lightweight, scalable bytecode-analysis tool that detects and classifies proxy contracts without relying on source code, transaction traces, or custom EVMs. By examining deployed bytecode and recursively following storage-based upgrade paths, PROXiFY flags both active proxies and dormant contracts whose upgrade mechanisms have never been triggered. Evaluation shows 98.6 % precision and 97.1 % recall. It is suitable for large-scale analysis and has been used to process 44 million contracts, identifying more than 1.3 million upgradeable proxies [7]. By reliably separating forward from upgradeable proxies, PROXiFY provides a practical foundation for improving transparency and safety in Ethereum-based DApps [8].

II. Methodology

PROXiFY is a tool designed to analyze Ethereum smart contract bytecode to determine whether a contract is an up-

[1]https://eips.ethereum.org/EIPS/eip-1967
[2]https://eips.ethereum.org/EIPS/eip-1822
[3]https://rekt.news/audius-rekt/
[4]https://github.com/IlhamQasse/PROXiFY

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00388

gradeable proxy, a forward proxy, or not a proxy. PROXiFY's methodology involves decompiling bytecode, detecting delegatecall, and recursively tracing the implementation address to classify proxy contracts. Figure 1 provides a high-level overview of this process, showing how the tool identifies and classifies proxies as forward or upgradeable.

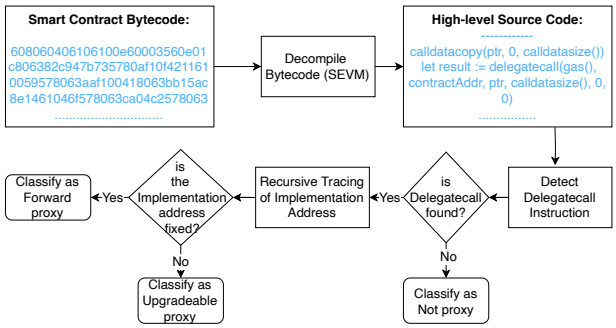

Fig. 1: High-level architecture of PROXiFY

A. Bytecode Decompilation

The first step involves decompiling the smart contract bytecode to obtain a human-readable representation of the contract's logic. PROXiFY takes the bytecode as input and uses a decompiler to translate it into an intermediate Solidity-like code. We selected SEVM,[5] an EVM decompiler, for its reliability and features. SEVM is lightweight, with no dependencies, and includes an embedded functions and events signature database. It can extract events or function information from bytecode and effectively identify the presence of `delegatecall` instruction.

B. Detection of `Delegatecall` Usage

One of the primary characteristics of a proxy contract is its use of the `delegatecall` instruction. `Delegatecall` allows a contract to call another contract's code while maintaining its own state. In proxy contracts, this is typically found in the fallback function, which forwards all unmatched calls to the implementation contract.

Listing 1: Fallback function performing `delegatecall`

```
address contractAddr = readAddress(IMPLEMENTATION_SLOT);
assembly {
 let ptr := mload(0x40)
 calldatacopy(ptr, 0, calldatasize())
 let result := delegatecall(gas(), contractAddr, ptr,
      calldatasize(), 0, 0)
}
```

As shown in Listing 1, the `readAddress()` function retrieves the implementation address from the `IMPLEMENTATION_SLOT`, but this operation only reads the address; it does not modify it. Therefore, while the fallback function forwards the call to the implementation contract, it does not indicate whether the implementation address can be modified.

[5]https://github.com/acuarica/evm

C. Recursive Tracing the Implementation Address

After detecting the use of `delegatecall`, the next step in PROXiFY's analysis is to trace the implementation address to determine whether it can be modified. This is crucial because upgradeable proxies rely on the ability to change the implementation address, while forward proxies do not allow such modifications. For example, based on Listing 1, PROXiFY must continue its analysis, searching for other functions that might be responsible for modifying the `IMPLEMENTATION_SLOT`.

In complex scenarios, proxy contracts may modify the implementation address through multiple layers of functions. For example, consider the following function, shown in Listings 2 that modifies the `IMPLEMENTATION_SLOT`.

Listing 2: Modifying the implementation address

```
function setImplementation(address _newImplementation)
     onlyProxyOwner public {
 setAddress(IMPLEMENTATION_SLOT, _newImplementation);
}
```

This function, `setImplementation()`, allows the proxy owner to change the implementation address by calling `setAddress()` with the `IMPLEMENTATION_SLOT` and the new implementation address. It is the first point where the `IMPLEMENTATION_SLOT` is modified, indicating that the proxy might be upgradeable.

The `setAddress()` function, shown in Listing 3, then passes the new address to `storageSet()`.

Listing 3: Passing the new implementation address to storage

```
function setAddress(bytes32 _key, address _value) internal
     {
 storageSet(_key, addressToBytes32(_value));
}

function storageSet(bytes32 _key, bytes32 _value) internal
     {
 assembly {
 sstore(_key, _value)
 }
}
```

The `sstore` instruction in the `storageSet()` function writes the new implementation address into the storage slot corresponding to `IMPLEMENTATION_SLOT`. This is the key operation that modifies the implementation address, signaling that the contract is upgradeable.

PROXiFY recursively traces these function calls, to determine whether the implementation address is being modified. When the `delegatecall` target is read from storage but that slot is never written inside the proxy, PROXiFY reads the slot value on-chain using `eth_getStorageAt`. If the

slot contains an address that points to deployed bytecode, PROXiFY fetches that bytecode and repeats the same analysis: check for `delegatecall`, locate the storage slot, and test whether that slot is writable. This recursive procedure follows upgrade paths that are one or more levels deep (e.g., Universal Upgradeable Proxy Standard (UUPS) or Beacon proxies). If the inspected slot is uninitialized (zero address), upgradeability cannot be confirmed, and the proxy is marked non-upgradeable.

D. Classification Logic

Once the analysis of function boundaries, `delegatecall`, and recursive tracing is complete, PROXiFY classifies the contract as one of the following:

- **Not a Proxy**: No `delegatecall` instruction is found.
- **Forward Proxy**: The implementation address is fixed and cannot be modified. For the example in Listing 1, If no modifications are found for the implementation address (either directly within the proxy or indirectly through the contract resolved from that slot), the contract is classified as a forward proxy, meaning the implementation cannot be changed.
- **Upgradeable Proxy**: The implementation address is stored in a modifiable state variable and can be updated through specific functions. In section II-C, Listing 3 illustrates that the implementation address is indeed being modified via functions like `setAddress()` and `storageSet()`, hence the contract is classified in this case as an upgradeable proxy.

E. Report Generation

After classification, PROXiFY outputs the classification and a detailed explanation based on the bytecode input. Users can download a report containing the results, which includes the contract address and the classification outcome. This feature allows users to integrate the tool into their workflows and analyze contracts efficiently.

III. Related Work

The detection and classification of proxy contracts in smart contracts have been approached using three primary methods: source code analysis, bytecode analysis, and a combination of bytecode and transaction history.

Source code analysis, as used by Bodell III et al. [9] and Liu et al. [12], provides a static view of proxy structures based on the availability of verified source code. While this approach offers detailed insights into proxy behavior, it is limited by the fact that a large proportion of smart contracts on Ethereum are unverified, making source code unavailable for analysis in many cases.

Bytecode-based analysis has addressed this limitation by removing the need for source code. Huang et al. [13] proposed a model that classifies upgradeable proxy contracts by analyzing opcode frequencies, achieving high accuracy. However, this model assumes that opcode frequency is a reliable indicator of upgradeability, which may not hold true in all scenarios. In some cases, the location of specific opcodes is more critical for detection than their frequency. Li et al. [14] extended bytecode analysis by incorporating predefined function selectors, such as `upgradeTo` and `setImplementation`, to detect common proxy patterns. While effective for known patterns, this approach may overlook unconventional proxy implementations that do not follow standard naming conventions. Proxion [15] combines symbolic execution with bytecode inspection to detect upgradeable proxies. While precise, this method requires a custom EVM setup and is computationally expensive, making it harder to scale to large datasets.

Lastly, combining bytecode analysis with transaction history has been explored to offer dynamic insights into contract behavior. Ebrahimi et al. [11] and Salehi et al. [8] use this combined method to detect upgradeable proxies. Although this approach provides more context on how contracts are used over time, it misses inactive proxies that have not yet undergone an upgrade, leaving parts of the proxy landscape undetected, where so far only 0.34% proxies were upgraded [7].

A. PROXiFY Evaluation

We evaluated PROXiFY using the ground truth dataset published by Ebrahimi et al. [16], which provides a refined classification of 916 contracts (653 upgradeable proxies and 263 non-upgradeable proxies). Our evaluation against this dataset produced the following results: 634 true positives, 244 true negatives, 9 false positives, and 19 false negatives. Based on these results, PROXiFY achieved **98.6% precision**, **97.1% recall**, and a **97.8% F1 score**.

a) Misclassification Analysis: We manually examined all 28 misclassified cases to understand the causes of failure. Among the false negatives, 12 contracts failed due to automation-related issues in the batch decompilation process. These were later verified manually and confirmed to be upgradeable, indicating that the decompiler or bytecode parser did not handle them correctly during bulk processing. The remaining seven contracts were misclassified due to inconsistencies introduced by the decompiler. Although these contracts follow standard upgrade patterns and modify the expected storage slot, the decompiled output showed a different slot being modified in the logic contract, which prevented our automated analysis from correctly resolving the upgrade path. In the nine false positive cases, the decompiled bytecode contained upgrade-related functions (e.g., `upgradeTo()`, `delegatecall()`) even though they were not present in the actual deployed bytecode or were never reachable. These functions appear to have been included in the decompiled output for completeness, leading to over-approximation during classification. The complete classification results are provided in the project repository.[6]

B. Limitations

The main limitation of PROXiFY lies in its dependence on decompilation quality, as misinterpretations in decompiled output caused most of the 28 misclassified cases in our evaluation.

[6]https://github.com/IlhamQasse/PROXiFY

While SEVM proved reliable for detecting `delegatecall` instructions during individual analyses, it occasionally introduced parsing errors in batch processing, leading to false negatives. Although PROXiFY currently uses SEVM as its default backend, its core logic is decompiler-agnostic. Other decompilers can be integrated as long as their output can be mapped to the same instruction and storage abstractions. We tested Heimdall[7] and Panoramix[8], but they either failed to detect `delegatecall` consistently or experienced timeout issues during processing.

IV. Applications of PROXiFY

PROXiFY offers multiple applications across research and practical development contexts, focusing on the upgradeability of Ethereum smart contracts.

First, researchers studying smart contracts' evolution or security implications can utilize PROXiFY to analyze datasets and classify proxies without requiring source code.By detecting active and inactive proxies, PROXiFY enables researchers to conduct empirical studies on upgradeability mechanisms, to investigate how smart contracts evolve and maintain state over time. This tool has already been used in our large-scale empirical study [7], which analyzed **44 million** smart contracts and identified over **1.3 million** upgradeable proxies.

For developers, PROXiFY ensures that the proxy contracts they implement function as intended. The tool can be used to verify if upgradeability mechanisms are correctly in place, helping developers avoid cases where upgrade functions are unreachable. It allows developers to confirm whether their smart contracts are upgradeable or fixed, ensuring their design intentions are reflected in the contract's behavior.

Security analysts and auditors can leverage PROXiFY's ability to detect upgradeable proxies directly from bytecode, even when source code is unavailable. This is particularly useful in ensuring that contracts, especially those in critical fields such as decentralized finance (DeFi), are correctly classified and adhere to standards like EIP-1967. By providing transparency into how the contract's logic may evolve, auditors can assess potential risks associated with the contract's upgradeability.

Lastly, individual users interacting with high-value contracts can use PROXiFY to assess whether the contracts they engage with are subject to change. Understanding whether a contract is upgradeable helps users make more informed decisions, particularly regarding the risks associated with evolving functionality, ensuring greater transparency in decentralized applications.

V. Conclusion

In this paper, we presented PROXiFY, a bytecode-based tool for detecting and classifying proxy contracts within Ethereum smart contracts. As the need for contract upgrades becomes increasingly essential in evolving decentralized applications, traditional detection methods are often insufficient, especially in identifying inactive or unverified proxies. By focusing on bytecode-level analysis, PROXiFY addresses key limitations of existing tools, enabling more coverage of both forward and upgradeable proxies without requiring source code or transaction history. Our evaluation shows that PROXiFY achieves 98.6 % precision, 97.1 % recall, and a 97.8 % F1 score on a manually validated benchmark of proxy contracts. This makes PROXiFY a valuable tool for developers, auditors, and researchers working within the Ethereum ecosystem. Future work includes optimizing the decompilation process for faster large-scale analysis.

[7]https://github.com/Jon-Becker/heimdall-rs.git

[8]https://github.com/eveem-org/panoramix

References

[1] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, and F.-Y. Wang, "Blockchain-enabled smart contracts: Architecture, applications, and future trends," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 49, no. 11, pp. 2266–2277, 2019.

[2] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, no. 37, pp. 2–1, 2014.

[3] D. He, Z. Deng, Y. Zhang, S. Chan, Y. Cheng, and N. Guizani, "Smart contract vulnerability analysis and security audit," *IEEE Network*, vol. 34, no. 5, pp. 276–282, 2020.

[4] M. Rodler, W. Li, G. O. Karame, and L. Davi, "{EVMPatch}: Timely and automated patching of ethereum smart contracts," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1289–1306, 2021.

[5] N. F. Samreen and M. H. Alalfi, "A survey of security vulnerabilities in ethereum smart contracts," *arXiv preprint arXiv:2105.06974*, 2021.

[6] P. Antonino, J. Ferreira, A. Sampaio, and A. Roscoe, "Specification is law: Safe creation and upgrade of ethereum smart contracts," *arXiv preprint arXiv:2205.07529*, 2022.

[7] I. Qasse, M. Hamdaqa, and B. Þ. Jónsson, "Immutable in principle, upgradeable by design: Exploratory study of smart contract upgradeability," *arXiv preprint arXiv:2407.01493*, 2024.

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

[9] 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 (USENIX Security 23)*, pp. 1829–1846, 2023.

[10] A. M. Ebrahimi, B. Adams, G. A. Oliva, and A. E. Hassan, "A large-scale exploratory study on the proxy pattern in ethereum," *Empirical Software Engineering*, vol. 29, no. 4, pp. 1–51, 2024.

[11] A. M. Ebrahimi, B. Adams, G. A. Oliva, and A. E. Hassan, "Upc sentinel: An accurate approach for detecting upgradeability proxy contracts in ethereum,"

[12] Y. Liu, S. Li, X. Wu, Y. Li, Z. Chen, and D. Lo, "Demystifying the characteristics for smart contract upgrades," *arXiv preprint arXiv:2406.05712*, 2024.

[13] Y. Huang, X. Wu, Q. Wang, Z. Qian, X. Chen, M. Tang, and Z. Zheng, "The sword of damocles: Upgradeable smart contract in ethereum," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, pp. 333–345, 2024.

[14] X. Li, J. Yang, J. Chen, Y. Tang, and X. Gao, "Characterizing ethereum upgradable smart contracts and their security implications," in *Proceedings of the ACM on Web Conference 2024*, pp. 1847–1858, 2024.

[15] C.-K. Chen, W.-Y. Chu, M. Tran, L. Vanbever, and H.-C. Hsiao, "Proxion: Uncovering hidden proxy smart contracts for finding collision vulnerabilities in ethereum," *arXiv preprint arXiv:2409.13563*, 2024.

[16] A. M. Ebrahimi, B. Adams, G. A. Oliva, and A. E. Hassan, "Upc sentinel: An accurate approach for detecting upgradeability proxy contracts in ethereum," *Empirical Software Engineering*, vol. 30, no. 3, pp. 1–79, 2025.