Investigating Technical Debt & Post-Deployment Evolution in Ethereum Smart Contracts: A Double-Edged Sword

by

Amirmohammad Ebrahimi

A thesis submitted to the Graduate Program in Computing

in conformity with the requirements for the

Degree of Doctor of Philosophy

Queen's University

Kingston, Ontario, Canada

December 2025

Copyright © Amirmohammad Ebrahimi, 2025

Abstract

BLOCKCHAIN technology facilitates the creation of decentralized, transparent, and tamper-resistant systems. Ethereum's immutable smart contracts underpin this trust but severely constrain post-deployment maintenance. This rigidity makes it difficult to address internal quality issues, including both unintentional bugs and deliberate compromises known as Technical Debt (TD). In this immutable context, TD poses unique risks as retroactive fixes are costly or infeasible. This thesis addresses this challenge through three interconnected studies.

First, we conduct a large-scale empirical analysis of Self-Admitted Technical Debt (SATD) in over 5.7 million verified smart contracts. Although only 1.5% of contracts contain SATD, they are prevalent in high-value applications handling significant transactions, suggesting an outsized impact on the ecosystem. Crucially, 93.5% are clones, indicating systematic propagation. We develop a domain-specific

taxonomy to categorize SATD, highlighting its practical relevance and systemic roots in Ethereum's environment.

Second, we analyze the architectural response to immutability: proxy contracts. We hypothesize these are widespread and conduct a large-scale study of over 50 million contracts. Our analysis reveals over 14% employ proxy patterns. We categorize them into Forwarder Proxy Contracts (67.8%), which delegate calls to a fixed logic contract for modularity and cost savings, and Upgradeable Proxy Contracts (32.2%), which enable post-deployment contract evolution. This significant use of UPCs demonstrates a growing reliance on upgradeable architectures.

Third, we focus on the heightened risks of UPCs, which introduce implementation complexity and centralization hazards. To address the limitations of existing detection methods, we introduce UPC Sentinel, a robust hybrid static-dynamic analysis tool that operates directly on bytecode. It accurately distinguishes UPCs from FPCs and classifies them into specific implementation variants. Our evaluation shows UPC Sentinel significantly outperforms state-of-the-art source-level detectors, enabling the reconstruction of version histories critical for assessing long-term maintainability.

Collectively, this work provides a holistic empirical and technical framework for understanding how Ethereum developers manage the tension between immutability and evolution. Our findings, implications, datasets, and methods empower developers, auditors, and researchers to reason more effectively about smart contract maintainability and evolution, laying a foundation for a more robust and sustainable Ethereum ecosystem.

Acknowledgments

I would like to express my deepest gratitude to Prof. Ahmed E. Hassan for giving me the opportunity to pursue my Ph.D. under his supervision. His guidance, encouragement, and unwavering support have been instrumental throughout my doctoral journey. I am truly fortunate to have worked with him and learned from his expertise, insightful feedback, and high standards of research. His mentorship not only shaped the direction of my work but also helped me grow as an independent thinker and researcher.

I would also like to sincerely thank Dr. Gustavo Ansaldi Oliva for patiently guiding me through a new line of research during my Ph.D. His thoughtful advice, attention to detail, and willingness to answer my many questions helped me navigate unfamiliar territory with confidence. I deeply appreciate his mentorship and the time he devoted to discussing ideas, reviewing drafts, and supporting my growth as a researcher.

I am especially grateful to Prof. Bram Adams for being not only a teacher and mentor but also a collaborator and a constant source of support and inspiration. His guidance extended far beyond technical advice, offering me the opportunity to gain valuable academic experiences. From serving as a Web Co-Chair for MSR 2025 to reviewing for top-tier journals and taking on instructor responsibilities, these experiences enriched my Ph.D. journey and broadened my perspective on academic service. I deeply appreciate his trust and encouragement throughout.

I would also like to dedicate this work to the memory of Prof. Ahmad Abdollahzadeh Barforoush, my MSc supervisor. I had the privilege of being his student and will always be grateful for his continuous support during my master's studies. His guidance extended beyond academics, helped me grow as an individual and inspired me through his integrity, kindness, and belief in my potential. His impact on my personal and academic development remains lasting and profound.

Last but not least, I am deeply grateful to my dad, mom, brother, and loving sister for their unwavering support, encouragement, and love throughout this journey. Their belief in me, even during the toughest moments, gave me the strength to keep going. This achievement would not have been possible without their constant presence and sacrifices.

Co-authorship

For each chapter and the related publications included in this thesis, my contributions include formulating the initial research ideas, reviewing background material and related work, collecting data, proposing the research methods, analyzing the results, and writing the manuscripts. My co-contributors supported this work by helping refine the initial research ideas, suggesting improvements to the research methods, and providing feedback on manuscript drafts. The work presented in this thesis has been submitted as listed below:

- A. M. Ebrahimi, G. A. Oliva and A. E. Hassan, Self-Admitted Technical Debt in Ethereum Smart Contracts: A Large-Scale Exploratory Study, in IEEE Transactions on Software Engineering, vol. 49, no. 9, pp. 4304-4323, Sept. 2023. This work is described under Chapter 4.

- Amir M. Ebrahimi, Bram Adams, Gustavo A. Oliva, and Ahmed E. Hassan. 2024. A large-scale exploratory study on the proxy pattern in Ethereum. Empirical Softw. Engg. 29, 4 (Jul 2024). This work is described under Chapter 5.

- Amir M. Ebrahimi, Bram Adams, Gustavo A. Oliva, and Ahmed E. Hassan. 2025. UPC sentinel: An accurate approach for detecting upgradeability proxy contracts in Ethereum. Empirical Softw. Engg. 30, 3 (May 2025). This work is described under Chapter 6.

Table of Contents

Abstract i

Acknowledgments iii

Co-authorship v

List of Tables x

List of Figures xi

Chapter 1: Introduction 1
1.1 Blockchain Technology 1
1.2 Immutability 2
1.3 Technical Debt 3
1.4 Smart Contract Evolution 6
1.5 Proxy Pattern 6
1.6 Detecting Upgradeable Proxy Contracts (UPCs) 9
1.7 Contributions 11
1.8 Thesis Structure 15

Chapter 2: Background 16
2.1 Blockchain 17
2.2 Ethereum 18
2.3 Smart Contracts 19
2.4 Decentralized Applications (DApps) 22
2.5 Immutability 22
2.6 Ether (ETH) 24
2.7 Account types 25
2.8 Transactions 26
2.9 Governance 28
2.10 EIPs & ERCs 30
2.11 Cross-Contract Interactions 30

2.12 Function Signatures & Selectors . . . 31
2.13 Panoramix Decompiler . . . 33
2.14 Smart Contract Evolution . . . 34
2.15 Proxy Pattern . . . 39

Chapter 3: Literature Survey . . . 60
3.1 Self-Admitted Technical Debt (SATD) . . . 61
3.2 Smart Contract Maintenance . . . 86
3.3 Proxy Contract Detection . . . 92

Chapter 4: Self-Admitted Technical Debt in Ethereum Smart Contracts: A Large-Scale Exploratory Study . . . 106
4.1 Introduction . . . 106
4.2 The Role of SATD in Smart Contracts . . . 110
4.3 Data Collection . . . 115
4.4 RQ1: How Prevalent Is SATD in Smart Contracts? . . . 119
4.5 RQ2: How Does the Prevalence of SATD in Smart Contracts Relate to Code Cloning? . . . 122
4.6 RQ3: What Types of SATD Do Smart Contracts Contain? . . . 132
4.7 Discussion . . . 145
4.8 Threats to Validity . . . 151
4.9 Conclusion . . . 153

Chapter 5: A Large-Scale Exploratory Study on the Proxy Pattern in Ethereum . . . 157
5.1 Introduction . . . 157
5.2 Data Collection . . . 164
5.3 Preliminary Study: Is the Proxy Pattern Prevalent Enough in Ethereum to Warrant Large-Scale Study? . . . 165
5.4 The Proposed Proxy Detector . . . 169
5.5 RQ1: How Prevalent Are Proxy Contracts in Ethereum? . . . 176
5.6 RQ2: What Are the Different Creational Patterns for Deploying Proxy Contracts? . . . 185
5.7 RQ3: To What Extent Do Proxy Contracts Differ in Type and Adhere to Established Reference Implementations? . . . 196
5.8 Discussion . . . 202
5.9 Threats to Validity . . . 226
5.10 Conclusion . . . 231

Chapter 6: UPC Sentinel: An Accurate Approach for Detecting Upgradeable Proxy Contracts in Ethereum . . . 236

6.1 Introduction . . . 236
6.2 UPC Sentinel . . . 243
6.3 Data Collection . . . 288
6.4 RQ1: How Accurately Does UPC Sentinel Detect Upgradeable Proxy Contracts in the GE Dataset? . . . 313
6.5 RQ2: How Does UPC Sentinel's Proxy Detector Compare to USCHUNT in Detecting Proxy Contracts? . . . 315
6.6 RQ3: How Does UPC Sentinel's Proxy Type Classifier Compare to USCHUNT in Classifying Proxy Contracts? . . . 319
6.7 RQ4: How Does UPC Sentinel's UPC Pattern Classifier Compare to USCHUNT in Classifying UPC Patterns? . . . 324
6.8 Discussion . . . 329
6.9 Threats to Validity . . . 345
6.10 Conclusion . . . 353

Chapter 7: Conclusion . . . 356
7.1 Scope . . . 356
7.2 Key Findings . . . 358
7.3 Thesis Contributions . . . 363
7.4 Practical Implications . . . 365
7.5 Limitations . . . 373
7.6 Future Work . . . 375
7.7 Closing Remarks . . . 382

Bibliography . . . 420

Appendix . . . 421
A.1 Evaluation of the Employed SATD Detector Tool . . . 421
A.2 Practical Applications of the Proxy Pattern in DApps . . . 422
A.3 Cost-Efficient Deployment with Proxy Contracts: A Case Study of Uniswap-V1 DApp . . . 424
A.4 Largest Group of Type-1 Contracts per Clone Cluster . . . 425
A.5 Examples of DApps for the Top-5 On-Chain Creational Patterns . . . 428

List of Tables

3.1 Overview of the candidate SATD detector methods (2014–2020). . . 85
3.2 Comparison of studies on proxy detection. Blank cells indicate missing or unspecified information in the corresponding studies. 96
3.3 Comparison of studies on proxy type classification. 102

4.1 Summary of the dataset used to train Jitterbug's easy-to-find component. 118
4.2 Contract deployments by account type in top-10 clusters. 128

5.1 The performance of our proxy detection approach. 174
5.2 Overview of the 12 detected proxy contract creational patterns sorted based on the category and the proxy instance count. 191
5.3 Distribution of use cases by deployment style (on-chain vs. off-chain) and use case size, measured by the number of proxy contracts per use case. 195

6.1 Comparison of implementation variable location and upgrade function location across UPC Patterns, analyzed at the source code (top) and bytecode (bottom) levels. Red-highlighted values indicate discrepancies between the two analysis levels (Bodell III et al. 2023). . . 250
6.2 Comparison of label distribution across Mixed and Active-Only settings.317
6.3 Comparison of UPC Sentinel's proxy detector and USCHUNT in Mixed and Active-Only settings. 318
6.4 Comparison of UPC and Non-UPC distributions across Mixed and Active-Only settings. 321
6.5 Comparison of UPC Sentinel's proxy type classifier and USCHUNT in Mixed and Active-Only settings. 323
6.6 Accuracy comparison between UPC Sentinel and USCHUNT across different UPC patterns. Asterisks (*) denote statistically significant differences based on McNemar's test. 328
6.7 Breakdown of UPC Sentinel's time cost across GE and GB datasets. . 345

A.1 Distribution of type-1 clones across the top 10 clone clusters. 426

List of Figures

1.1 The smart contract development, deployment, and evolution lifecycle, with thesis chapters mapped to their respective areas of research focus. . . . 4
1.2 Evolution of proxy pattern standards for smart contracts. . . . 8

2.1 Trace table for example transaction *T*. . . . 29
2.2 Call graph of example transaction *T*. . . . 29
2.3 Structural UML diagram of the GoF proxy design pattern . . . 40
2.4 Key components of the Inherited Storage UPC Pattern. . . . 51
2.5 Key components of the Eternal Storage UPC Pattern. . . . 52
2.6 Key components of the Registry UPC Pattern. . . . 53
2.7 Key components of the Beacon UPC Pattern. . . . 54
2.8 Key components of the ERC-1967 UPC Pattern. . . . 55
2.9 Key components of the ERC-1822 Universal UPC Pattern (UUPS). . . . 56
2.10 Key components of the ERC-2535: Diamonds UPC Pattern. . . . 58
2.11 Key components of the Transparent UPC Pattern. . . . 59

4.1 An overview of our data collection process. . . . 115
4.2 Distributions of activity level (left) and currency influx (right). Red lines denote the 95th percentile. . . . 121
4.3 Cumulative coverage of clone clusters in terms of the ratio of SATD contracts. . . . 125
4.4 Evolution of the top-10 largest clone clusters. . . . 125
4.5 Correlation heatmap between SATD count and contract size. . . . 131
4.6 Our classification of SATD types found in the studied smart contracts. Each box shows a specific subtype with its number of occurrences. . . . 137

5.1 Monthly ratio of multi-contract transactions. . . . 169
5.2 Monthly ratio of multi-contract transactions by call type. . . . 169
5.3 Trace table of example transaction *T*. . . . 170
5.4 CCDFs of the number of logic contracts per proxy. . . . 182
5.5 CCDFs of the number of inbound transactions. . . . 182

5.6 The count of EOAs deployed contracts as of each month. 183
5.7 Ratio of EOAs who deployed a proxy as of each month. 183
5.8 Monthly number of distinct use cases. 185
5.9 Monthly ratio of transactions that involve at least one proxy. 185
5.10 The monthly ratio of multi-contract transactions that involve at least one proxy contract. 186
5.11 Example of sequence of contract creations. 188
5.12 Example of labeled sequence of contract creations. 188
5.13 Metamodel of the 12 creational patterns for proxy contract deployment.190
5.14 CCDF of the number of proxy contracts per use case, comparing on-chain and off-chain deployment styles. 193
5.15 CCDF of the number of proxy contracts per use case, comparing the top-5 on-chain creational patterns with the off-chain pattern. 193
5.16 Distribution of proxy use cases by reference implementation type. . . 201
5.17 Decision flowchart for selecting between off-chain and on-chain deployment styles. 221

6.1 UPC Sentinel architecture. 245
6.2 Illustrative example of our proxy detector leveraging its two inherent properties. 246
6.3 Structure of the three Upgradeable Proxy Contract Reference Designs (UPC Reference Designs) and their respective variants. Note that all identifiers shown (e.g., function names and variable names) are illustrative; arbitrary identifiers may be used in practice. 253
6.4 Detailed design of the "SMUP Detector" component. 258
6.5 Three bytecode features illustrating the two conceptual properties of proxy contracts (Section 2.15.4) at the decompiled bytecode level. . 260
6.6 Detailed design of the "ESUP Detector" component. 264
6.7 Detailed design of the "DUP Detector" component. Small diamonds without decision criteria represent UML flowchart merge nodes. . . . 268
6.8 Detailed design of the "Auxiliary Impact Variable Detector" module. . 270
6.9 Detailed design of the "Upgrade Function Detectors" module. 271
6.10 Detailed design of the "Contract Bytecode Decompiler" module. . . . 272
6.11 A systematic top-down approach for identifying the type of upgradeable proxy pattern a UPC adheres to. 275
6.12 Monthly deployment trend of the 3,177 Upgradeable Proxy Contracts (UPCs) collected from Etherscan. 291
6.13 Protocol for examining the ground truth contracts with respect to Property #1 (i.e., the presence of `delegatecall`). 301

6.14 The protocol for examining the ground truth contracts against Property #2 (proxy must maintain the interface of the object it represents). The three patterns illustrating this property are highlighted in yellow. 304
6.15 Protocol for examining the ground truth contracts against Property #2 (i.e., the proxy must maintain the interface of the object it represents). The three patterns illustrating this property are highlighted in yellow. 307
6.16 Confusion matrices for proxy detection by UPC Sentinel and USCHUNT in Mixed and Active-Only settings. 319
6.17 Confusion matrices for proxy type classification by UPC Sentinel and USCHUNT in Mixed and Active-Only settings. 324
6.18 Simplified class diagram of the `ProxyDeployer` contract. Contracts with gray backgrounds represent upgradeable proxy contracts. 330
6.19 Simplified class diagram of the `AddressUtils` contract. Contracts with gray backgrounds represent dead upgradeable proxy contracts. . 333
6.20 Simplified class diagram of the `GEOPAY` contract. Contracts with gray backgrounds represent upgradeable proxy contracts. 336
6.21 Recovering a contract's release history using UPC Sentinel. Highlighted colors indicate the locations where the upgrade function selector and distinct implementation contract addresses are identified within the retrieved transactions. 340

CHAPTER 1

Introduction

1.1 Blockchain Technology

BLOCKCHAIN technology has emerged as a transformative force in various sectors (Miraz and Ali 2018; Chen et al. 2018b; Gartner 2021) by enabling decentralized, transparent, and tamper-resistant systems (Swan 2015). A programmable blockchain is a decentralized system that serves as both an immutable database and a distributed world computer, where autonomous programs can be deployed and executed across a network of computational computers, eliminating the need for centralized intermediaries and fostering trustless environments.

Ethereum is the most widely used open-source, programmable blockchain platform. Unlike Bitcoin, which primarily supports financial transactions, Ethereum enables the execution of smart contracts (Szabo 1994; Wood 2014), allowing developers to build sophisticated blockchain-powered applications, known as decentralized applications (DApps) (Oliva et al. 2020). These DApps span various domains, including decentralized finance (DeFi) platforms such as Uniswap and Aave, non-fungible token (NFT) marketplaces like OpenSea, blockchain-based games such as Axie Infinity, and decentralized autonomous organizations (DAOs) like Aragon.

Smart contracts are self-executing and Turing-complete software programs that enforce agreements when specific conditions, written in the code and stored on the blockchain, are met. Their execution on the Ethereum network is governed by the Ethereum Virtual Machine (EVM), a runtime environment that ensures deterministic behavior across all nodes by interpreting the contract's compiled bytecode. The top-half of Figure 1.1 shows a typical workflow of developing and deploying a smart contract. Once the code is written, it is compiled into bytecode, which is the format the EVM understands. This bytecode is then packaged into a deployment transaction using tools like Hardhat or Remix and sent to the immutable Ethereum blockchain, where the EVM executes it on every node. As these contracts operate autonomously and are **immutable by design**, correctness and security are critical from the outset.

1.2 Immutability

Immutability is a fundamental property of the Ethereum blockchain. Once a smart contract is deployed, its bytecode is permanently stored and cannot be modified,

a property enforced by Ethereum's consensus mechanism (Section 2.2) to ensure tamper-proof behavior. However, this immutability becomes a significant liability when a contract contains a deficiency, whether unintentional or intentional (e.g., vulnerabilities, bugs, technical debt), as developers cannot patch the code after deployment (Chen et al. 2021c). History is replete with severe consequences arising from this rigidity. For example, the DAO attack exploited a bug in an immutable contract, resulting in the loss of approximately $50M (Mehar et al. 2019). Similarly, a vulnerability in the Parity multi-signature wallet led to roughly $150M being permanently locked due to the contract's immutable design (Suiche 2017).

In response to these financial risks, researchers have extensively focused on identifying and preventing unintentional vulnerability patterns (Luu et al. 2016; Kalra et al. 2018; Chen et al. 2018a; Tikhomirov et al. 2018; So and Oh 2023; Chu et al. 2023). However, significantly less attention has been paid to intentional weaknesses, particularly technical debt (TD) (Bosu et al. 2018; Chen et al. 2021c). Figure 1.1 provides an overview of the smart contract lifecycle, encompassing the initial development and deployment phases, followed by the processes of identifying post-deployment deficiencies and executing patches through evolution mechanisms. The figure also illustrates how the main published chapters of this thesis contribute to and integrate with this lifecycle.

1.3 Technical Debt

TD is a software engineering (SE) metaphor for the trade-off where developers choose a quick or suboptimal implementation instead of a better solution, incurring

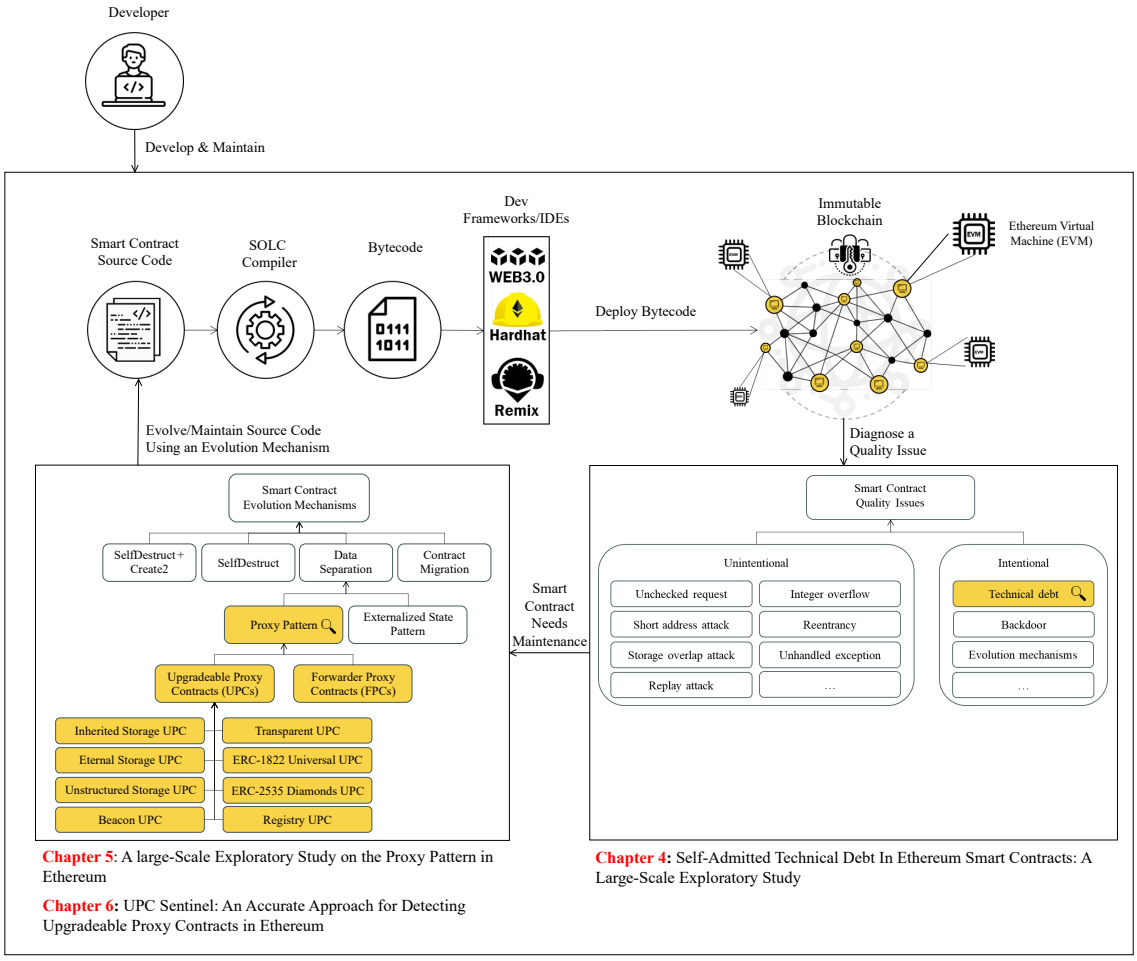


Figure 1.1: The smart contract development, deployment, and evolution lifecycle, with thesis chapters mapped to their respective areas of research focus.

a "debt" of additional work to be handled later Cunningham 1992; Potdar and Shihab 2014; Sierra et al. 2019; Kashiwa et al. 2022; Vidoni 2021; Xiao et al. 2022. In the context of smart contracts, such debt may manifest as omitting essential features, relying on ad-hoc workarounds, deferring optimizations, or providing insufficient documentation to expedite deployment under tight deadlines or uncertain specifications.

We conjecture that factors such as fear of missing out (Porru et al. 2017; Labazova et al. 2019; Oliva et al. 2020; Gartner 2021), time-to-market pressure (Chakraborty et al. 2018), immature tooling (e.g., specialized IDEs, programming languages, debuggers, testing frameworks, package managers), and Ethereum's evolving roadmap (Becze and Jameso 2015) often compel or force developers to prioritize rapid delivery over maintainability and long-term correctness, thereby contributing to the accumulation of technical debt (TD) in smart contracts.

Although prior surveys suggest that technical debt is a common concern among blockchain developers (Bosu et al. 2018; Chen et al. 2021c), no large-scale empirical study has yet validated the extent to which this debt manifests in deployed smart contracts. While taking on TD can accelerate development in the short term, the immutable nature of Ethereum smart contracts makes it significantly harder and costlier to repay that debt through later modifications. This amplifies the impact of suboptimal design decisions and underscores the need to understand TD in smart contracts not merely as a code quality issue, but as a fundamental risk in blockchain applications.

Hence, as the first large-scale empirical study, we aim to demystify the characteristics of technical debt in smart contracts. Understanding these characteristics helps reveal intentional shortcomings in deployed code, uncover their origins, highlight their implications for stakeholders, and inform potential strategies for avoidance or mitigation. In doing so, our study complements prior research that has focused primarily on unintentional deficiencies by examining another critical source of deficiencies: those arising from intentional, conscious trade-offs that developers make under pressure. These trade-offs are often overlooked, yet they are

equally significant to the long-term maintainability of blockchain applications (Cunningham 1992; Potdar and Shihab 2014).

1.4 Smart Contract Evolution

Whether the result of intentional or unintentional deficiencies, smart contracts require effective evolution mechanisms to address internal quality issues, adapt to changing requirements, and incorporate improvements (Lehman et al. 1997; Bosu et al. 2018; Lohr and Peldszus 2020; Chen et al. 2021c). As mentioned earlier, unlike traditional centralized software, which benefits from iterative development and patching, Ethereum lacks built-in support for contract evolution due to the inherent immutability of the blockchain. Consequently, developers must proactively embed evolution mechanisms into the smart contract architecture prior to deployment to enable effective post-deployment maintenance.

Over time, the Ethereum community has devised various mechanisms to support contract evolution, including contract migration, low-level EVM instructions such as `selfdestruct` and `create2`, and data separation (Section 2.14). Among these, the "proxy pattern", which is a specific class of data separation, has emerged as the primary mechanism, not only enabling contract evolution but also improved modularity, enhanced code reuse, and reduced operational costs.

1.5 Proxy Pattern

In a proxy setup, an immutable "proxy smart contract" at a fixed address receives user calls and subsequently seamlessly delegates them to a separate predefined

"logic smart contract" that contains the actual functionalities. Proxy contracts generally fall into two categories (Jorge Izquierdo 2018; Santiago Palladino 2019b): i) Upgradeable Proxy Contracts (UPCS), which allow an authorized entity to update the proxy's reference to a new logic contract, thereby enabling the evolution of the logic contract; and ii) Forwarder Proxy Contracts (FPCs), which delegate calls to a fixed hard-coded logic contract often to promote code reuse, reduce operational costs, and support modular design (Mudge 2020a; Peter Murray 2018a). The popularity of this approach is theoretically evident in Figure 1.2, which illustrates the progression of proposed standards and updates to the proxy pattern introduced by the Ethereum community.

Despite their benefits, proxies also represent a double-edged sword by undermining contract transparency and predictability (Antonino et al. 2022), introducing centralization risks (Palladino 2020; Salehi et al. 2022), and exposing new attack surfaces and vulnerabilities (Bodell III et al. 2023; Zhang et al. 2025b). More specifically, since proxy contracts mediate between users and the underlying logic contract, they introduce an additional layer of indirection that can obscure the actual logic running behind the scenes, reducing transparency (Jorge Izquierdo 2018; Santiago Palladino 2019b). This indirection, combined with the ability to modify the logic contract post-deployment, can introduce centralization risks and enable malicious behavior without user awareness (Antonino et al. 2022; Ethereum 2023b; Pathade and Hooli 2025). This problem is compounded by the fact that up to 99% of Ethereum contracts are closed-source (Xu et al. 2020; Huang et al. 2021; Chen et al. 2019; Chen et al. 2022a), leaving only bytecode available for analysis, which further hinders auditing (Center 2022).

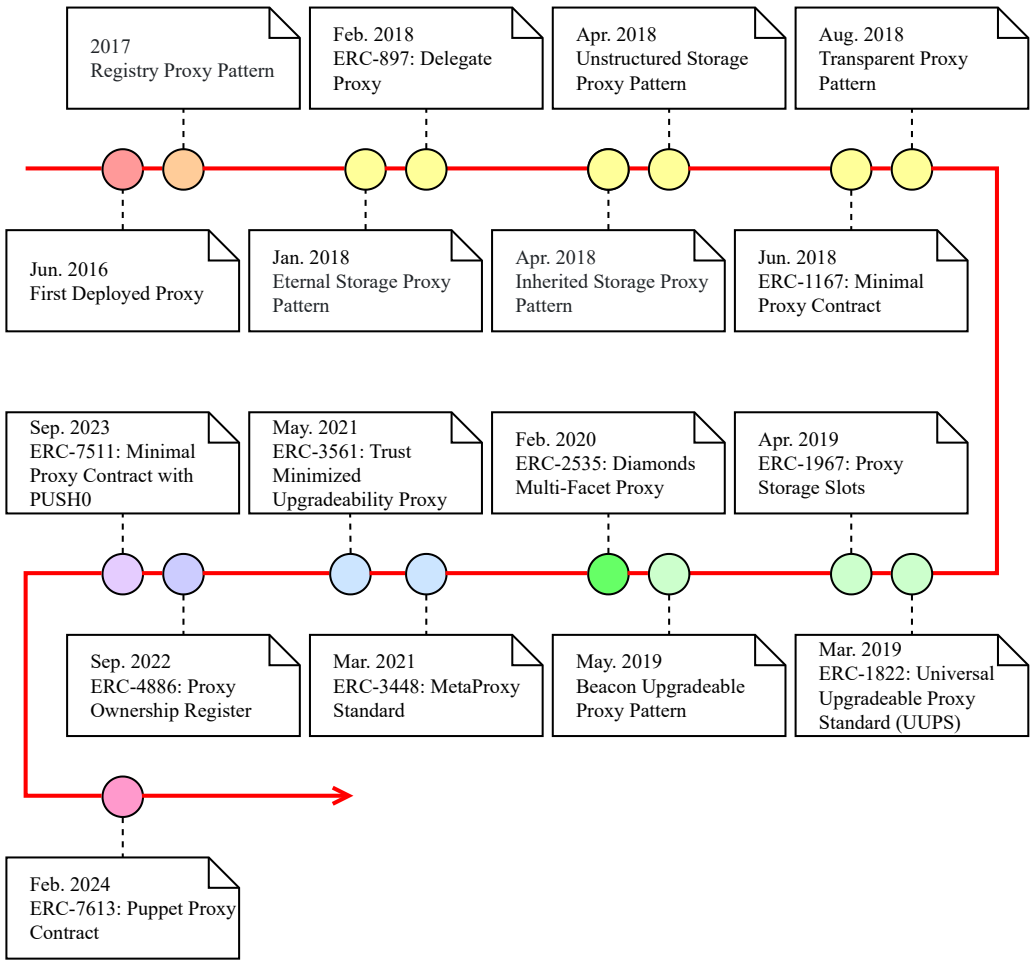


Figure 1.2: Evolution of proxy pattern standards for smart contracts.

Hence, detection and analysis of proxy contracts is crucial. For end users, flagging proxy contracts enhances awareness and facilitates timely notifications about changes to the underlying logic contracts. Practitioners, such as auditors, also benefit, as it enables them to identify proxy contracts and their associated logic contracts when assessing potential vulnerabilities and attack scenarios (Feist et al.

2019; Salehi et al. 2022; Bodell III et al. 2023; Ruaro et al. 2024; Chen et al. 2024; Zhang et al. 2025a; Huang et al. 2024).

While we hypothesize, given the many years of evolution of the proxy pattern (Figure 1.2), that proxy contracts are remarkably prevalent in practice, significant gaps remain in accurately detecting proxy contracts at scale and in understanding their real world prevalence, integration strategies, and the distribution of different implementation variants.

To validate our hypothesis, in our second large-scale empirical study, we first propose **a high-performing method that detects proxy contracts and all their associated logic contracts through mining historical transactions.** We then use this method characterize proxy contracts at scale. By filling these critical detection and characterization gaps, our study lays the groundwork for the final chapter, which builds on this foundation to accurately identify the type of a proxy contract, specifically distinguishing between UPCs and FPCs.

1.6 Detecting Upgradeable Proxy Contracts (UPCs)

Determining whether a given proxy contract is a UPC or an FPC involves analyzing whether the proxy offers interfaces, which typically correspond to upgrade functions through which an admin can update the reference to the logic contract, that enable post-deployment evolution. However, making this determination is not trivial. First, detection is complicated by the fact that most Ethereum contracts are closed-source, with only their bytecode publicly available, rendering source-based analysis impractical. Second, the growing diversity of UPC variants, now exceeding eight known versions, adds further complexity, as these variants differ both

structurally and behaviorally. Third, effective detection requires in-depth analysis of cross-contract interactions, combined with careful orchestration of static and dynamic analysis techniques. Despite these challenges, we hypothesize that it is feasible to accurately determine the type of proxy contract directly at the bytecode level.

Another key limitation is Ethereum's lack of a native version control system. Unlike traditional software, where developers rely on systems such as Git to track code changes, maintain version histories, and manage branches, Ethereum does not have built-in mechanisms to trace how a contract evolves over time. Once a contract is deployed, its state and bytecode are fixed on the blockchain, and any updates require deploying a new contract or using proxy patterns. However, there is no standard way to formally link these versions or document what changes were made between them. This absence of version control complicates efforts to track changes across contract versions and makes it difficult to analyze the history of design decisions, refactoring, or bug fixes. As a result, researchers and auditors face significant barriers when trying to study the long-term evolution of smart contracts, assess how technical debt (TD) accumulates or is repaid, and understand the lifecycle of security vulnerabilities. In short, the lack of version control creates a visibility gap in the software evolution process on the blockchain.

To address these gaps, **we propose UPC Sentinel, a method capable of precisely classifying a contract as UPC or FPC and identifying the specific UPC variant at the bytecode level.** We empirically evaluate UPC Sentinel against the current state-of-the-art detection method, USCHUNT, which identifies upgradeable

proxy contracts at the source code level. Our method also enables automated detection of smart contracts that are evolvable through the proxy pattern mechanism and paves the way for reconstructing their release histories. This provides a more accurate and comprehensive foundation for understanding trends in contract evolution and their broader social and ecosystem-level impacts in future studies.

In summary, our study is guided by the following hypotheses:

H1. Technical debt is significantly common in Ethereum smart contracts.

H2. Proxy contracts constitute a significant proportion of deployed Ethereum smart contracts.

H3. Despite challenges such as closed-source code, diverse UPC variants, and complex cross-contract interactions, it is feasible to accurately distinguish Upgradeable Proxy Contracts (UPCs) from Forwarder Proxy Contracts (FPCs) and their variants directly at the bytecode level.

1.7 Contributions

1.7.1 H1

To validate our first hypothesis (H1), we present the first in-depth, large-scale empirical study of technical debt (TD) in Ethereum smart contracts (Chapter 3.1), in which we made the following contributions:

We provide the first comprehensive empirical study of TD in Ethereum smart contracts, contextualizing its expression and consequences within the blockchain ecosystem, particularly focusing on the impact of immutability. Our research offers

empirical evidence that TD is actively incurred in deployed smart contracts, including high-profile projects like CryptoKitties and OpenSea, confirming that TD is not merely theoretical but a practical issue in the smart contract landscape.

Moreover, we identify a significant connection between the spread of TD and code cloning, demonstrating that technical debt is not only introduced by developers but also propagated through automatic mechanisms within the smart contract ecosystem. A detailed taxonomy of TD specific to Ethereum smart contracts is developed, grounded in rigorous qualitative studies and tailored to the unique constraints of the Ethereum ecosystem.

Additionally, we discuss key open research challenges and practical implications for managing TD in smart contracts, emphasizing how specific Ethereum features, such as the lack of patching capabilities, high upgrade costs, and the extensive use of proxies for code reuse, impact remediation strategies. By addressing gaps in the literature, we analyze over 5.7 million verified smart contracts, evaluating the applicability of state-of-the-art TD detection techniques in this domain. This research contributes essential data and insights to a field that lacks specialized software engineering practices, curated datasets, and actionable guidance for developers, researchers, and tool makers such as Etherscan (Bosu et al. 2018; Chen et al. 2021c).

1.7.2 H2

To validate our second hypothesis (H2), we present the first large-scale exploratory study of proxy contracts in Ethereum since the platform's inception, in which we made the following contributions:

We introduce the first large-scale exploratory study on proxy contracts in Ethereum, providing an in-depth analysis of their prevalence, trends, and deployment practices. Our study proposes a precise and scalable behavior-based method for detecting active proxy contracts, achieving a perfect score with 100% precision and recall. Through this method, we quantify the prevalence of proxy contracts and identify temporal trends, revealing their widespread and increasing use across more than 50 million deployed contracts and 1.6 billion transactions.

Additionally, we identify and analyze 12 distinct creational patterns for deploying proxy contracts, which are categorized into two dominant deployment styles: on-chain and off-chain. We assess the popularity, trade-offs, and practical considerations for selecting among these patterns, offering guidelines to facilitate informed decision-making in the development of Ethereum contracts.

We employed both quantitative and qualitative methods to classify proxy contracts according to their implementation variants (such as ERC-1167 and ERC-1967) and intended purposes (e.g., Upgradeable Proxy Contracts (UPCs) vs. Forwarder Proxy Contracts (FPCs)), respectively, uncovering initial trends in the adoption of different design choices. To support future research, we release four ground truth datasets covering both proxy detection and proxy type classification, ensuring reproducible evaluation in future studies.

We also analyze the use of proxy contracts in real-world decentralized applications (DApps), offering valuable insights into how they enable contract evolution, modularity, cost reduction, and logic reuse. While exploring the benefits, we also discuss the risks and trade-offs associated with proxy contracts, including the challenges of reduced transparency, governance centralization, and increased attack

surfaces. Lastly, we identify key open research challenges and articulate the practical implications of the proxy pattern, addressing concerns related to transparency, auditability, contract evolution analysis, and governance risks.

1.7.3 H3

To validate our third hypothesis (H3), we introduce UPC Sentinel, a robust and accurate hybrid static-dynamic analysis algorithm designed for detecting and classifying proxy smart contracts at the bytecode level (Chapter 6). Building on the foundational insights from our first large-scale study of proxy contracts in Ethereum (Chapter 5), UPC Sentinel follows a three-layered design that comprises: (i) a proxy detector to identify active proxy contracts, as identified in H2, (ii) a proxy type classifier to distinguish between Upgradeable Proxy Contracts (UPCs) and Forwarder Proxy Contracts (FPCs), and (iii) a UPC pattern classifier that performs fine-grained identification of the specific UPC pattern employed. The design of UPC Sentinel is grounded in a principled approach that incorporates three canonical reference architectures for upgradeable proxy contracts, derived from the taxonomy of upgradeable patterns curated by Meisami and III (2023).

We also release a new, high-quality ground truth dataset containing over 3,177 verified UPCs, and we identify and rectify labeling inconsistencies in a previously released dataset by Bodell III et al. (2023), which contains over 994 proxy contracts. Our extensive empirical evaluation of over 4,000 labeled contracts compares UPC Sentinel with the state-of-the-art source code-level detection tool, USCHUNT (Bodell

III et al. 2023). We demonstrate that UPC Sentinel outperforms USCHUNT by operating effectively in closed-source scenarios and generalizing to a wider variety of UPC variants.

Additionally, we highlight how UPC Sentinel can facilitate deeper future analysis of smart contract evolution, especially for closed-source contracts, surpassing existing methods in both the depth and breadth of its detection capabilities. Finally, we release UPC Sentinel and its associated datasets publicly to support reproducibility and facilitate community-wide benchmarking.

1.8 Thesis Structure

The remainder of this thesis is organized as follows: Chapter 2 introduces essential background concepts related to Ethereum, smart contracts, and upgradeability mechanisms such as the proxy pattern. Chapter 3 surveys prior work on technical debt detection and smart contract maintenance, summarizing key research gaps. Chapter 4 presents the first large-scale empirical study of technical debt in Ethereum smart contracts, examining its prevalence, characteristics, and implications. Chapter 5 details a second empirical study that introduces a scalable, behavior-based approach to detecting proxy contracts. Chapter 6 introduces *UPC Sentinel*, a hybrid static–dynamic analysis algorithm for detecting and classifying upgradeable proxy contracts (UPCs) directly at the bytecode level, along with a comprehensive evaluation against state-of-the-art tools. Finally, Chapter 7 concludes the thesis by summarizing key contributions, discussing practical implications, and outlining promising avenues for future research on smart contract maintainability and evolution.

CHAPTER 2

Background

In this chapter, we establish the foundational concepts and technical underpinnings relevant to the rest of the thesis. It begins with an overview of blockchain technology and programmable blockchains, focusing on Ethereum as the primary platform of study. We then introduce key Ethereum components, including its consensus mechanisms, account structure, transaction model, and gas system. Special attention is given to the Ethereum Virtual Machine (EVM), smart contracts, and the role of immutability in decentralized applications. We explore how smart contracts are created, interacted with, and verified, including mechanisms for function dispatch and cross-contract communication. Tools like Etherscan and the Panoramix decompiler are also discussed for reverse engineering and analyzing deployed contracts. The latter sections of the chapter focus extensively on

smart contract evolution: various contract evolution mechanisms, with a focus on the proxy design pattern, and a taxonomy of Upgradeable Proxy Contract (UPC) patterns.

2.1 Blockchain

A blockchain is a decentralized, distributed ledger that records transactions across a network of computers, known as nodes (Tripathi et al. 2023). Data is stored in sequential groups called blocks, each of which contains a set of transactions and a cryptographic reference to the previous block, forming an immutable chain. This chained structure ensures that altering any block would require recalculating all subsequent blocks and gaining consensus from the network, making the system highly secure and tamper-resistant. Nodes work together to maintain a consistent view of the blockchain through a consensus mechanism, which determines how new blocks are validated and added to the chain. The blockchain's design fosters transparency, trust, and decentralized control over data and transactions (Abdi et al. 2025).

A programmable blockchain is an extension of this concept, where, in addition to recording transactions, users can deploy and execute custom code directly on the blockchain network. This capability allows for the automation of complex processes, making programmable blockchains particularly useful for smart contracts and decentralized applications (DApps), where business logic and rules are encoded directly into the blockchain's infrastructure.

2.2 Ethereum

Ethereum is the first programmable blockchain platform that incorporates a global, shared computer known as the Ethereum Virtual Machine (EVM), embedded within its network. In Ethereum, all nodes maintain a copy of the EVM's state and collectively agree on its changes through a consensus mechanism, currently Proof-of-Stake (PoS), which replaced Proof-of-Work (PoW) in 2022 to improve energy efficiency and introduce a more sustainable and economically aligned security model (Ethereum 2025b). The state of Ethereum and all Etherscan is a widely used blockchain explorer and analytics platform for the Ethereum network. It provides a user-friendly interface to access detailed information about Ethereum transactions, wallet addresses, smart contracts, tokens, and more.

The consensus mechanism is a critical stack of protocols, incentives, and concepts that allows a decentralized network of nodes to achieve agreement on the state of a distributed ledger, without relying on a central authority. A robust consensus mechanism ensures that, despite the potential presence of faulty or malicious actors, at least a super majority (such as 66% of Ethereum validators) consistently agrees on the blockchain's global state. Ethereum's PoS mechanism relies on validators who stake cryptocurrency (e.g., ETH) as collateral and are randomly selected to propose and attest to new blocks. Honest validators earn rewards for correctly validating blocks, while those who act maliciously or go offline face penalties, including slashing, where they lose part of their staked funds.

The Ethereum Virtual Machine (EVM) is responsible for executing smart contract bytecode. Developers write smart contracts in high-level languages like Solidity, which are then compiled into bytecode that the EVM processes. When a

user submits a transaction that interacts with a smart contract, the EVM executes the corresponding code, leading to state changes that are immutably recorded on the blockchain. A key feature of Ethereum is its support for smart contracts, self-executing programs deployed on the EVM. This makes Ethereum not only a decentralized ledger but also a platform for running decentralized applications (DApps) that automate complex processes through smart contract execution.

2.3 Smart Contracts

A smart contract is a self-executing Turing-complete program deployed on the Ethereum blockchain that facilitates, verifies, or enforces the negotiation or performance of an agreement without the need for intermediaries. Once deployed, smart contracts are immutable, meaning their code cannot be altered or patched, underscoring the importance of thorough testing and verification prior to deployment (Zheng et al. 2020).

Smart contracts are predominantly written in Solidity, a statically typed, contract-oriented programming language with syntax reminiscent of JavaScript. Upon compilation, Solidity code is transformed into bytecode, which is then stored on the Ethereum blockchain. Notably, the source code itself is not stored on-chain, rendering the deployed bytecode opaque to human readers. Smart contracts form the foundation of decentralized applications (DApps), enabling trustless, automated interactions across various sectors such as finance (DeFi), supply chain management, governance, and digital identity. In the remainder of this thesis, the term smart contract will be used to refer to deployed contracts.

A verified smart contract on Ethereum is one whose source code has been publicly uploaded and matched against the on-chain bytecode by a blockchain explorer like Etherscan. This verification process ensures that the human-readable source code corresponds exactly to the compiled code deployed on the blockchain.

To illustrate the basic structure and functionality of a smart contract, consider the following example of a simple event ticketing system. This contract, shown in Listing 2.1, consists of two Solidity contracts: the base contract `AdminControlled` (Lines 2–13), and the application-specific contract `EventTickets` (Lines 14–36). The `AdminControlled` contract encapsulates administrator logic, including an `onlyAdmin` modifier that restricts function access to the deploying address. This logic is inherited by `EventTickets`, which uses it to control administrative privileges such as ticket issuance. The derived contract also defines a state variable `eventName`, a mapping `ticketHolders` that assigns ticket IDs to owners, and emits a `TicketIssued` event when a new ticket is created.

This example demonstrates several key features of smart contract programming in Solidity: state variables (Lines 16–18), access control via modifiers (Line 9), event logging (Line 20), and object-oriented design through inheritance (Line 15). It highlights how developers can encapsulate reusable logic (e.g., administrator control) in a base contract and extend it in specialized, application-level contracts.

```
1  pragma solidity ^0.8.0;
2  // Base contract for admin access control
3  contract AdminControlled {
4      address public admin; // Address with admin rights
5      constructor() {
6          admin = msg.sender; // Set deployer as admin
7      }
8      // Restricts access to admin only
9      modifier onlyAdmin() {
10         require(msg.sender == admin, "Not authorized");
11         _;
12     }
13 }
14 // Contract for issuing and managing event tickets
15 contract EventTickets is AdminControlled {
16     string public eventName; // Name of the event
17     uint256 public ticketCount; // Tracks issued ticket count
18     mapping(uint256 => address) public ticketHolders; // Maps ticket ID to owner
19
20     event TicketIssued(uint256 ticketId, address indexed to); // Log when ticket is
           issued
21
22     constructor(string memory _eventName) {
23         eventName = _eventName; // Set event name on deployment
24     }
25     /// @notice Admin issues a ticket to a recipient
26     function issueTicket(address to) public onlyAdmin {
27         require(to != address(0), "Invalid recipient");
28         uint256 ticketId = ++ticketCount; // Increment and assign ticket ID
29         ticketHolders[ticketId] = to; // Assign ticket to recipient
30         emit TicketIssued(ticketId, to); // Emit event
31     }
32     /// @notice Get the owner of a specific ticket
33     function getTicketOwner(uint256 ticketId) public view returns (address) {
34         return ticketHolders[ticketId];
35     }
36 }
```

Listing 2.1: An example of a smart contract for event ticketing in Solidity.

2.4 Decentralized Applications (DApps)

A decentralized application (DApp) typically consists of three main components: (1) a blockchain-based backend (usually Ethereum), (2) smart contracts deployed on the blockchain, and (3) a frontend user interface (UI), often implemented as a web or mobile application. The frontend communicates with the smart contracts through an interface layer (commonly using libraries like Web3.js or Ethers.js) that connects to the blockchain network via a node (e.g., Infura or Alchemy). The blockchain layer ensures secure and immutable execution of smart contract logic and maintains the state of the application. Together, these components allow users to interact with the DApp through a familiar web interface, while the underlying transactions and state changes are transparently recorded and secured by the blockchain network. DApps span a wide range of domains, including decentralized finance (DeFi), gaming, digital collectibles, decentralized autonomous organizations (DAOs), metaverse applications, and decentralized storage (Qasse et al. 2020; Etherscan 2023a).

2.5 Immutability

Immutability is a foundational property of the Ethereum blockchain, referring to the fact that once data, including smart contract code, is deployed on-chain, it cannot be modified or deleted (Zheng et al. 2020). This design stems from the decentralized and trustless nature of blockchain systems: immutability ensures that all participants can verify and rely on a consistent history of transactions and contract

logic without depending on a central authority. In Ethereum, key facets that are immutable include the deployed smart contract bytecode and the transaction history recorded on the blockchain. However, the internal state of a contract (i.e., the value of storage variables), can be changed over time in response to user interactions or on-chain events. This distinction between immutable code and mutable state allows contracts to maintain dynamic behavior, such as updating balances, managing ownership, or reacting to external inputs, while ensuring that the underlying logic governing these changes remains fixed and auditable.

Immutability in Ethereum is enforced at two levels: the EVM prevents alteration of deployed contract code through its execution model, while the consensus mechanism ensures all nodes maintain agreement on the unchangeable transaction and contract history. This rigidity brings important benefits, such as enhanced security, auditability, and resistance to tampering or censorship, but also presents significant challenges for contract evolution and long-term maintainability. Bugs, design flaws, technical debt, or logic updates cannot be patched directly, leading developers to adopt architectural workarounds (i.e., contract evolution mechanisms), such as the proxy pattern, to enable post-deployment contract evolution (a.k.a. contract upgrades). However, such patterns introduce new complexities and risks, including increased gas costs, upgrade-related vulnerabilities, centralization risks, and governance overhead. Thus, while immutability strengthens trust and transparency in decentralized applications, it necessitates careful planning and foresight in contract design to accommodate future changes.

2.6 Ether (ETH)

Ether (ETH) is the native cryptocurrency of the Ethereum blockchain, playing multiple roles within the ecosystem: as a medium of exchange, a store of value, and, most critically, as the currency used to pay for transaction fees and computational services (gas) (Oliva and Hassan 2021). Every operation on the Ethereum network, including deploying smart contracts and executing transactions, requires gas fees paid in ETH.

In the Ethereum network, every transaction incurs a fee known as "gas", which compensates validators for the computational resources required to process and validate operations (Zarir et al. 2021). This mechanism not only incentivizes validators but also serves as a deterrent against network spam by attaching a cost to each operation. The total gas fee for a transaction is determined by two primary components: the amount of gas used and the gas price.

- **Gas Used**: This refers to the total units of gas consumed by the transaction's operations. Each operation within the Ethereum Virtual Machine (EVM) has a predefined gas cost, reflecting the computational effort required. For instance, a simple ETH transfer typically consumes 21,000 gas units, while more complex operations, such as interacting with smart contracts, may require significantly more gas.
- **Gas Price**: Denominated in gwei (where 1 gwei = 10^{-9} ETH), the gas price represents the amount a user is willing to pay per unit of gas. Users can set this price based on their urgency; higher gas prices can expedite transaction processing by making them more attractive to validators.

The total transaction fee is calculated as:

$$\text{Total Fee} = \text{Gas Used} \times \text{Gas Price}$$

2.7 Account types

In Ethereum, an account is an entity that can send transactions and has a balance (measured in ether, ETH). It's essentially a user or a smart contract that can interact with the Ethereum network. Ethereum defines two primary types of accounts: externally owned accounts (EOAs) and contract accounts (CAs) (Ethereum 2025a; Etherscan 2023b). Both account types share common fields:

- **Address**: A 20-byte (40 hexadecimal characters) unique identifier prefixed with `0x`.
- **Nonce**: A counter that tracks the number of transactions sent from the account (for EOAs), or the number of contracts created by the account (for CAs). It ensures each transaction is processed exactly once and in order, preventing replay attacks.
- **Balance**: The amount of Ether (ETH) held by the account, denominated in **wei**, the smallest unit of ETH (1 ETH = 10^{18} wei).
- **CodeHash**: For contract accounts, this is the hash of the account's deployed bytecode, which defines the contract's logic. For EOAs, this field is the hash of an empty string.

- **StorageRoot**: A hash of the root node of a Merkle Patricia Trie that encodes the storage contents of the account. This data structure allows efficient verification of stored key-value pairs in the contract's persistent state. For EOAs, the storage is empty by default.

EOAs are controlled by users who hold the corresponding private keys. Creating an EOA costs nothing. Only EOAs can initiate transactions, which may transfer ETH/tokens to other EOAs, or call functions on contract accounts. EOAs do not contain executable code and do not maintain on-chain storage beyond the fields listed above. CAs; however, are created when a user deploys a smart contract to the network. This requires gas, as it consumes network storage. Contract accounts do not have private keys. Instead, they are controlled entirely by their deployed code. A contract account cannot initiate transactions on its own, but can execute its code and send internal transactions in response to receiving a transaction or message call (Etherscan 2023b). Contracts can perform a wide range of operations, such as transferring ETH or tokens, interacting with other contracts, or even deploying new contracts.

2.8 Transactions

In Ethereum, a transaction is a cryptographically signed instruction initiated by an externally owned account (EOA) to effect a change in the network's state. Transactions are the primary mechanism for interacting with the Ethereum blockchain(solidity

2023). Each transaction includes fields such as: sender's address, recipient's address (if applicable), ETH value transferred, `calldata` payload, gas limit, gas price, and a nonce to prevent replay attacks. They can take several forms:

- **Ether transfer transactions**: sending ETH from one EOA to another.
- **Contract deployment transactions**: deploying new smart contracts to the blockchain. These transactions do not have a to address; instead, the `calldata` field contains the compiled contract bytecode.
- **Contract interaction transactions**: invoking functions on already-deployed smart contracts. In this case, the recipient's address is the address of the target contract, and the `calldata` field specifies the target function selector and its corresponding encoded arguments.

While only EOAs can initiate transactions, smart contracts can interact with other contracts through internal transactions, commonly referred to as "traces". These traces represent operations triggered during the execution of a transaction, such as function calls between contracts. Although not actual transactions recorded on the blockchain, traces provide a detailed view of the interactions and state changes that occur during contract execution. Each trace includes information about the operation type, involved addresses, input and output data, value transferred, gas used, and the execution result.

Figure 2.1 illustrates the trace table for an example transaction T[1]. For better readability, we labeled the externally owned account (EOA), the involved contracts, and each trace with appropriate abbreviations and color coding in the figure. As

[1] View Etherscan transaction

shown, the EOA initiates a transaction by calling contract *C1*, resulting in three internal traces: *T1*, *T2*, and *T3*. Specifically:

- In *T1*, contract *C1* creates a new contract, *C2*.
- In *T2*, *C1* calls *C2*.
- In *T3*, triggered by *T2*, *C2* delegates a call to another contract, *C3*.

This demonstrates that traces can form a hierarchical structure, where a single trace can have one or more child traces depending on the contract logic and execution flow. Figure 2.2 presents the corresponding call graph for transaction *T*, showing the precedence relationships among the traces. The directed edges in the graph indicate the parent-child hierarchy. For instance, trace *T3* is a child of *T2*, indicating that *T2* precedes and initiates *T3*.

Lastly, when a contract is deployed by an EOA, it typically means that a user or developer has sent a transaction to the network where the contract bytecode is included in the `data` field of the transaction. In contrast, when a contract *C2* is deployed by another contract *C1*, it implies that *C1* invoked the `CREATE` or `CREATE2` opcode during its execution as part of a transaction *T*. In such cases, the bytecode for *C2* is either hardcoded within *C1* or constructed dynamically at runtime, and then used to deploy *C2*.

2.9 Governance

Governance refers to the mechanisms and processes through which decisions are made, implemented, and enforced in decentralized systems (OpenZeppelin 2023a).

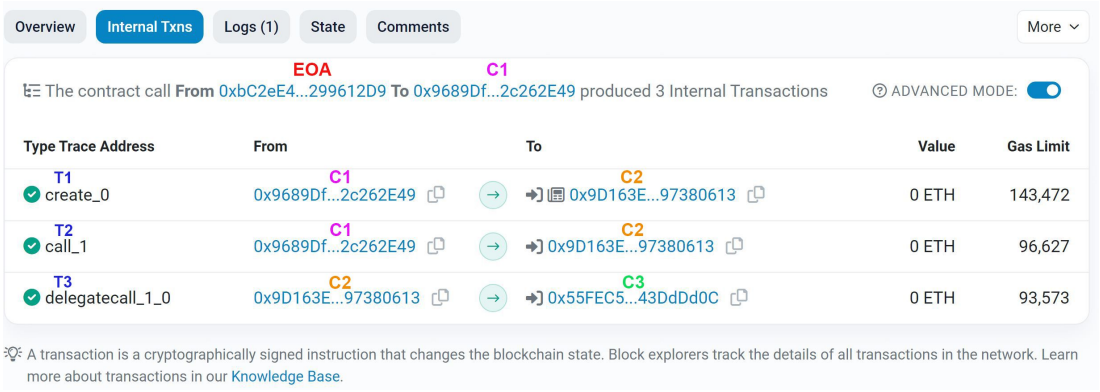


Figure 2.1: Trace table for example transaction T.

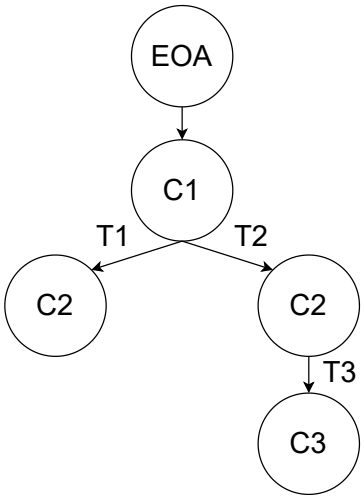


Figure 2.2: Call graph of example transaction T.

In the context of Ethereum smart contracts, governance often involves how contract upgrades, parameter changes, and critical administrative actions are managed post-deployment. Unlike traditional software, smart contracts are immutable by default, making pre-defined governance mechanisms, such as voting schemes, multi-signature approvals, and role-based permissions, essential for adaptability and control. Contracts lacking proper governance structures may become obsolete,

vulnerable, or misaligned with stakeholder interests over time. Thus, effective governance is a key factor in the long-term sustainability and security of decentralized applications.

2.10 EIPs & ERCs

The evolution of the Ethereum platform is guided through a formal process of community-driven proposals known as Ethereum Improvement Proposals (EIPs) (Becze and Jameso 2015). An EIP is a design document that outlines new features, technical changes, or process updates intended to improve the Ethereum protocol. EIPs are publicly reviewed and debated by the community, core developers, and stakeholders before potential adoption. A subset of EIPs is classified as Ethereum Request for Comments (ERCs), which specifically propose application-level standards, such as token specifications and contract interface conventions. For instance, ERC-20 defines a standard interface for fungible tokens, while ERC-721 introduces a standard for non-fungible tokens (NFTs). The EIP/ERC process fosters transparency, community participation, and interoperability across the Ethereum ecosystem.

2.11 Cross-Contract Interactions

Solidity supports four types of cross-contract calls: `call`, `callcode`, `staticcall`, and `delegatecall` (Patra 2024). To clarify their differences, consider three contracts: Alice, Bob, and Charlie. When Alice uses `call` to invoke Bob, the code executes in Bob's context, and Bob's storage may be modified. Executing "in the

context of a contract" means the code runs using the target contract's storage, balance, and address, with all state changes applied to that contract's storage. The `staticcall` instruction, introduced as a security enhancement, enables contract calls while strictly prohibiting any state modifications during the call or its subcalls. If Alice calls Bob, and Bob in turn uses `callcode` to invoke Charlie, Charlie's code executes in Bob's context, allowing Charlie to modify Bob's storage. In this scenario, Charlie perceives Bob as the message sender. However, `delegatecall` improves upon `callcode` by preserving the original caller: if `callcode` is replaced with `delegatecall`, Charlie sees Alice, rather than Bob, as the message sender. This distinction makes `delegatecall` the preferred method for scenarios requiring execution in the caller's context while preserving the original call chain.

2.12 Function Signatures & Selectors

In EVM, the mechanism for invoking functions within a smart contract relies on *function signatures* and *function selectors*. These constructs are central to how Ethereum handles function dispatch. A function signature is a canonical representation of a function's name and its input parameter types (Marx 2018). It is written in the format:

```
functionName(type_1,type_2,...,type_n)
```

For example, the Solidity function:

```
function transfer(address recipient, uint256 amount)
```

has the function signature:

```
transfer(address,uint256)
```

Note that the parameter names (`recipient` and `amount`) are not included in the function signature, only the types are considered. This string uniquely identifies the function's interface, excluding return types, modifiers, and visibility specifiers.

To efficiently route a function call, the EVM uses its function selector, which is the first 4 bytes (8 hexadecimal characters) of the Keccak-256 hash of the function signature. The function selector serves as a unique identifier for the function within the contract's bytecode. This selector is embedded in the transaction's `calldata`, which encodes the function call and its arguments. Specifically, the first four bytes of the `calldata` represent the function selector, followed by the ABI-encoded parameters. For instance, the selector for `transfer(address recipient, uint256 amount)` is computed as:

```
keccak256(transfer(address,uint256)) = 0xa9059cbb...
```

Taking the first 4 bytes of the hash yields the selector `0xa9059cbb`. When a transaction is submitted to a contract, the EVM reads these four bytes from the `calldata` to determine which function to invoke. If a matching selector is found in the contract's dispatch table, the call is routed to the corresponding function implementation. If no match is found, the contract's `fallback` function (if defined) is executed instead. These special functions act as catch-all handlers for unmatched function selectors or plain Ether transfers. Notably, the `fallback` function does not have a function selector and is invoked when no other function in the contract matches the provided selector.

2.13 Panoramix Decompiler

Panoramix is a decompiler designed to reverse-engineer Ethereum smart contract bytecode (Palkeo 2019). Integrated into Etherscan, the most widely used blockchain explorer, Panoramix offers a high-level view of smart contract internals by translating bytecode into a flattened, linearized representation. This process resolves inherited functions and internal calls, making the contract logic easier to analyze. Listing 2.2 presents an excerpt from the decompiled bytecode of a real-world proxy contract[2]. Typically, decompiled contracts consist of two segments: (i) the storage section (lines 3–8), which defines variables at the contract level, and (ii) the business logic (lines 10–42), which contains public and external functions used for interaction.

In this example, lines 10–14 define getter functions that return the admin and implementation addresses. The fallback function, spanning lines 16–24, performs the proxy operation: line 18 shows a `delegatecall` to the implementation address stored in the `stor3608` variable (i.e., the implementation variable), while line 19 confirms that the function selector is derived from the first four bytes of the calldata, indicating compliance with the secondary proxy property. Lines 26–36 define the `upgradeTo` functions, which update the implementation address. Finally, the `changeAdmin` function (lines 32–36) allows the admin address of the proxy to be modified.

[2] 0xee6a57ec80ea46401049e92587e52f5ec1c24785

2.14 Smart Contract Evolution

Smart contract evolution, also referred to as contract upgrades, is the act of modifying or extending a smart contract's behavior after deployment to address new requirements or correct deficiencies. These deficiencies may arise from security vulnerabilities, logical bugs, or technical debt that was either unintentionally introduced or intentionally deferred due to time-to-market constraints.

Ethereum smart contracts are immutable by design (Section 2.5). This immutability ensures transparency and trust in decentralized applications. However, it also means that the bytecode deployed at a contract's address cannot be changed after deployment. To reconcile the need for both immutability and maintainability, developers have devised a variety of smart contract evolution mechanisms, also known as upgradeability mechanisms, which are design strategies that enable contracts to evolve in a controlled and secure manner (Palladino 2020). In the following sections, we explain various contract evolution mechanisms.

2.14.1 Parameterization

This mechanism evolves contract behavior by adjusting predefined configuration parameters (e.g., a fee rate) without altering the core logic. It is simple and transparent but offers a limited scope for change, as it cannot fix bugs or add features. Major risks include misconfiguration and values falling outside safe operational bounds, necessitating built-in validation logic and safeguards to prevent failures.

```
# Panoramix decompiler.

def storage:
  stor3608 is uint128 at storage 0x360[...]bbc offset 160
  stor3608 is addr at storage 0x360[...]bbc
  stor3608 is uint256 at storage 0x360[...]bbc
  storB531 is uint128 at storage 0xb531[...]103 offset 160
  storB531 is addr at storage 0xb531[...]103

def admin():
  [...]

def implementation():
  [...]

def _fallback() payable:
  [...]
  delegate uint256(stor3608.field_0) with:
    funct call.data[0 len 4]
     gas gas_remaining wei
    args call.data[4 len calldata.size - 4]
  if not delegate.return_code:
     revert with ext_call.return_data[0 len return_data.size]
  return ext_call.return_data[0 len return_data.size]

def upgradeTo(address _implementation):
  require calldata.size - 4 >= 32
  [...]
  addr(stor3608.field_0) = _implementation
  [...]

def changeAdmin(address _admin):
  require calldata.size - 4 >= 32
  [...]
  addr(storB531.field_0) = _admin_
  [...]
```

Listing 2.2: An example of decompiled bytecode obtained using the Panoramix decompiler.

2.14.2 Contract Migration

Migration involves deploying a new contract at a new address and manually transferring state from the old one. It is used for major fixes or when no upgrade mechanism exists. The process is labor-intensive, error-prone, and costly, requiring state recovery (complex for mappings) and batched writes due to gas limits. A critical challenge is user migration, as all users and integrated services must be redirected to the new address, creating significant coordination overhead and security risks.

2.14.3 Strategy Pattern

This design pattern enables limited upgradeability by encapsulating specific algorithms into separate, interchangeable "strategy" contracts. The main contract delegates certain operations to these satellites by holding a mutable reference to their address. This allows for partial, modular upgrades without a full system change. Its key limitation is that it cannot fix vulnerabilities in the main contract's core logic and is best suited for localized, infrequent behavior changes.

2.14.4 Self-Destruct

`selfdestruct` is a low-level EVM instruction that enables a smart contract to delete its own bytecode from the blockchain and transfer its remaining Ether to a specified address. Due to its irreversible nature and potential for abuse (e.g., rug pulls), it poses severe security risks and is formally deprecated, making it an obsolete and dangerous tool for contract upgrades (Entriken 2022).

2.14.5 Metamorphic Create2

This advanced technique uses the `CREATE2` opcode to deploy new contract logic to the same predetermined address (Fröwis and Böhme 2022). A factory deploys a logic contract and a lightweight "metamorphic" contract that copies the logic. To upgrade, the metamorphic contract is destroyed and redeployed with new logic. While it preserves the contract address for users, it is inherently risky as it relies on the deprecated `selfdestruct` opcode and is vulnerable to malicious takeover, leading to its phase-out.

2.14.6 Data Separation

To reduce the cost and risk of migrating smart contract state, data separation mechanisms (Palladino 2020; Meisami and III 2023; Salehi et al. 2022; Li et al. 2024c) decouple business logic from persistent storage by placing them in distinct contracts. This architectural principle enables more flexible contract upgrades by allowing developers to evolve logic without rewriting or migrating user data. We categorize data separation approaches into three main types: basic data separation, satellite-based separation, and the proxy pattern.

2.14.6.1 Basic Data Separation

A single logic contract interacts with a single storage contract. Upgrading logic requires a new deployment but the storage remains intact, eliminating data migration. Users must still be moved to the new logic contract address.

2.14.6.2 Satellite-Based Separation

Extends the basic model using multiple modular satellite contracts for specific logic or data segments. Upgrades involve deploying a new satellite and updating a reference pointer. This allows fine-grained control but introduces complexity in managing references, ensuring interface compatibility, higher gas costs, and fragmented data.

2.14.6.3 Proxy Pattern

The most advanced form of data separation involves proxies, which maintain a fixed entry point (proxy contract) while delegating execution to a logic contract using low-level instructions like `delegatecall`. In this pattern, the proxy holds all state variables, and logic resides in a separate implementation contract. Upgrades are performed by modifying the reference to the implementation address, often through admin-controlled functions. This approach enables seamless user interaction and interface continuity across upgrades. However, it introduces increased complexity, particularly in ensuring consistent storage layouts across implementations and mitigating security risks associated with `delegatecall`. We further elaborate on this mechanism in Section 2.15.

While these mechanisms avoid full data migration, they each introduce coordination and security challenges. Developers must enforce strict access controls to prevent unauthorized logic contracts from manipulating the storage, ensure interface compatibility across upgrades, and manage the gas overhead introduced by inter-contract communication.

2.15 Proxy Pattern

The proxy design pattern is one of the structural patterns cataloged by the Gang of Four (Gamma et al. 1994) in their seminal work Design Patterns: Elements of Reusable Object-Oriented Software. As a structural pattern, the proxy is primarily concerned with the composition of classes and objects to form larger structures. The core intent of the Proxy pattern is to provide a surrogate or placeholder for another object to control access to it. Figure 2.3 shows that the proxy design pattern involves three core components:

- Common: An interface that defines the common operations for both the Logic and the Proxy.
- Logic: The actual object that implements and performs real operations.
- Proxy: Maintains a reference to the Logic and implements the Subject interface. It may perform access control, resource management, or optimization before delegating the request to the Logic.

Structurally, the proxy pattern introduces an intermediary Proxy that implements the same interface as the Logic it represents. In other words, the Proxy preserves the interface of the Logic. This ensures that clients remain unaware whether they are dealing with the Logic or its Proxy, allowing proxies to be seamlessly interchanged with their logic from the perspective of the client (Shapiro 1986).

Inspired by the classic proxy pattern introduced in the Gang of Four (GoF) design patterns (Section 2.3), the proxy pattern in smart contracts enables indirect interaction between users (or external contracts) and core contract logic. In the context of Ethereum, promotes modularity, supports contract evolution, improves

separation of concerns, and can reduce operational costs. Smart contract setups that implement this pattern are commonly referred to as *proxy smart contracts*.

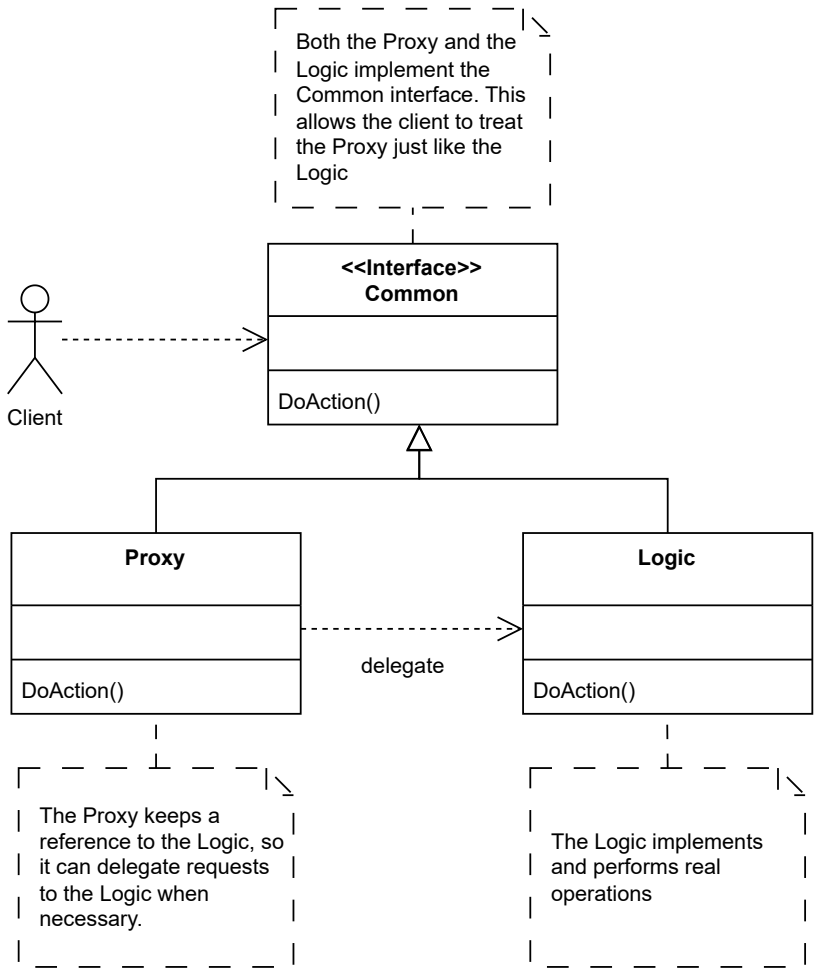


Figure 2.3: Structural UML diagram of the GoF proxy design pattern

2.15.1 Core Components

- **Proxy Contract**. Listing 2.3 shows an example code snippet of a proxy contract. The proxy contract serves as the single entry point for all external interactions with the application. These interactions include any calls initiated by users, other smart contracts, or decentralized applications (DApps) attempting to invoke functions or query data. Rather than executing logic itself, the proxy intercepts these calls and redirects them to a designated logic (implementation) contract using a low-level EVM instruction called `delegatecall` (lines 24). This forwarding mechanism ensures that the external caller interacts with the application through a stable contract address (the proxy), even if the underlying logic is evolved over time. The proxy is also responsible for maintaining the application's persistent state, including a state variable that keeps track of the current logic contract address (lines 5). In most designs, a `fallback` function is used to catch and delegate all unmatched function calls (lines 19–32), ensuring seamless redirection to the logic contract.

- **Logic (Implementation) Contract**. Listing 2.3 shows an example code snippet of a logic contract, implementing a simple wallet application. The logic contract (a.k.a., implementation contract) contains the core logic and function definitions that define the behavior of the application, such as updating state variables, performing computations, enforcing access control, and responding to user inputs. Unlike traditional contracts, the logic contract is not directly invoked by users or external systems. Instead, it is designed to be accessed exclusively via the proxy contract. At runtime, when a call is delegated by the proxy using `delegatecall`, the logic contract's code is executed

as if it were part of the proxy. As a result, all state changes, including variable updates and balance adjustments, affect the proxy's storage, not the storage of the logic contract itself. This design allows developers to update, evolve or replace the logic contract over time without disrupting the persistent state held in the proxy (lines 5–6 for layout alignment).

2.15.2 State Management

In proxy-based smart contract systems, one of the most critical design considerations is how state is managed and shared between the proxy and its associated logic contract. This is governed by the behavior of the low-level `delegatecall` instruction, which executes the logic contract's code in the context of the proxy (Listing 2.3, line 24). When a function is invoked through the proxy, `delegatecall` ensures that all operations, such as reading from or writing to storage variables, are performed using the proxy's storage, not the storage of the logic contract itself. As a result, while the logic contract may define state variables, it does not maintain its own persistent state, all reads and writes are directed to the proxy's storage during execution. It simply defines logic that manipulates the proxy's storage layout. Due to this shared state context, the storage layout in the logic contract must be carefully aligned with that of the proxy contract (Listing 2.3, lines 5–6; Listing 2.4, lines 5–6). Developers must avoid defining storage variables independently in the

```
pragma solidity ^0.8.0;

contract Proxy {
    // Storage slot 0 for the address of the current logic (implementation) contract
    address public implementation; // slot 0
    address public admin;  // slot 1

    constructor(address _logic) {
        implementation = _logic;
        admin = msg.sender;
    }
    // Function to update the logic contract address
    function upgradeTo(address newImplementation) external {
        require(msg.sender == admin, "Only admin can upgrade");
        require(newImplementation != address(0), "Invalid address");
        implementation = newImplementation;
    }
    // Fallback function catches all calls and forwards them to the logic contract
    fallback() external payable {
        assembly {
            // Copy msg.data to memory
            calldatacopy(0, 0, calldatasize())
            // Perform delegatecall
            let result := delegatecall(gas(), implementation, 0, calldatasize(), 0, 0)
            // Copy returned data
            returndatacopy(0, 0, returndatasize())
            // Return or revert based on result
            switch result
            case 0 { revert(0, returndatasize()) }
            default { return(0, returndatasize()) }
        }
    }
}
```

Listing 2.3: An example of a proxy contract.

logic contract without coordinating them with the proxy's storage layout. Otherwise, mismatches can lead to storage collisions, where unrelated variables overwrite each other's data during `delegatecall` operation, resulting in critical bugs or security vulnerabilities (Section 5.8.1.2).

```
pragma solidity ^0.8.0;

contract WalletLogic {
    // Must match storage layout of proxy: slot 0 and 1
    address public implementation; // Slot 0
    address public admin;   // Slot 1

    modifier onlyOwner() {
        require(msg.sender == owner, "Not authorized");
        _;
    }
    // This function can be called via proxy to set up state
    function initialize(address _owner) public {
        require(owner == address(0), "Already initialized");
        owner = _owner;
    }
    // This function can be called via proxy to set up state
    function initialize(address _admin) public {
        require(admin == address(0), "Already initialized");
        admin = _admin;
    }
    // Logic for depositing funds
    function deposit() public payable { }
    // Logic for withdrawing funds
    function withdraw(uint256 amount) public onlyAdmin {
        payable(msg.sender).transfer(amount);
    }
    // Logic for transferring funds by the admin
    function transfer(address recipient, uint256 amount) public onlyAdmin {
        payable(recipient).transfer(amount);
    }
}
```

Listing 2.4: An example of a logic contract that implements a wallet application.

2.15.3 Interface Preservation

In the conventional Gang of Four (GoF) Proxy Design Pattern (Section 2.15), the proxy and real logic (implementation) both explicitly implement the same common interface, allowing them to be used interchangeably. This structural symmetry

enables the proxy to stand in for the real object transparently, typically adding pre/post-processing logic (e.g., access control, caching, logging) while forwarding calls.

In Ethereum's proxy pattern, the proxy contract also maintains or preserves the interface of its logic contract, though it does so with a few key implementation differences. More specifically, the proxy contract and the logic contract typically do not implement the same common interface. That is, the proxy contract typically does not directly define any of the actual functions exposed by the logic contract. Instead, it relies on a built-in Solidity mechanism known as the `fallback()` function (Listing 2.3, lines 19–32) to dynamically handle and forward incoming calls. The `fallback()` function is automatically triggered whenever a call to the contract does not match any of the explicitly defined function signatures in the proxy. This behavior is fundamental to how the proxy exposes an interface that mirrors that of the logic contract without explicitly declaring it.

In a proxy setup, when triggered, the `fallback()` function retrieves the address of the current logic contract (typically stored in a reserved state variable; Listing 2.3, line 5) and forwards the raw calldata, containing the function selector and arguments, using `delegatecall` (Listing 2.3, line 24). This forwarding operation does not alter the `calldata` or execution context, allowing the logic contract to interpret the function call and execute its corresponding code as if it had received the call directly.

From the caller's perspective, whether it is an end user, a DApp, or another contract, the proxy appears to expose the same interface as the logic contract. This is

possible because Ethereum clients and tools like Web3.js or ethers.js format function calls based on the logic contract's function signatures and structure. When these encoded calls are sent to the proxy, the `delegatecall` function ensures they are forwarded and executed on the correct logic contract. Thus, while the Solidity source code of the proxy does not explicitly define any application logic functions, the combination of `fallback()` and `delegatecall` enables the proxy to dynamically reflect the interface of its underlying logic contract.

The `fallback()` function plays a crucial role in proxy contracts by reducing code duplication and enabling a minimal, generalized design. Instead of redeclaring every function from the logic contract, the proxy uses a single `fallback()` function to catch and forward all external calls using `delegatecall`. This approach keeps the proxy lightweight and agnostic to the specific logic being executed, allowing it to remain unchanged even as the underlying logic contract evolves. By abstracting the delegation mechanism, the `fallback()` function decouples the proxy from the logic contract's interface, ensuring that contract evolution can be performed simply by updating the stored address of the logic contract, without requiring modifications to the proxy contract source code itself.

2.15.4 What Makes a Contract a Proxy?

Based on established guidelines and standards (Gamma et al. 1994; Agarwal 2023; Wasnik 2024; Mudge 2020a; Fravoll 2018; OpenZeppelin 2017), we identify two *inherent and conceptual properties* that characterize proxy contracts. Let P be a contract under analysis. We consider P to be a proxy contract if and only if it satisfies the following two properties:

- **Property #1.** P must *expose* at least one public or external function, denoted F_P, through which it forwards calls to a corresponding function F_L in a separate logic contract. This delegation is performed using the low-level `delegatecall` instruction. The function F_P is typically the `fallback()` function of P (Bodell III et al. 2023; Salehi et al. 2022; Qasse et al. 2023; OpenZeppelin 2017), though any explicitly defined forwarding function may serve the same role if `fallback()` is not used.

- **Property #2.** Assuming Property #1 holds, the function selector of F_P must match exactly with that of F_L. This requirement ensures that P faithfully preserves the interface of the logic contract it represents (Gamma et al. 1994; Agarwal 2023; Wasnik 2024). During execution, `delegatecall` uses the first four bytes of the `calldata` (i.e., the function selector of F_P) to resolve and execute the corresponding function F_L in the logic contract (Mudge 2020a; Fravoll 2018). If this interface compatibility is not satisfied, the contract P is more accurately classified as an *adapter contract* rather than a proxy (Gamma et al. 1994).

In other words, both the proxy and logic contracts must have at least one function with a matching selector, and the proxy contract must employ the `delegatecall` to forward calls within this function to the corresponding function in the logic contract.

2.15.5 Types of Proxy Contracts

While all proxy contracts share the foundational mechanics and components, not all proxies are created with the same design intentions or operational capabilities.

Broadly, proxy contracts can be categorized into two distinct types based on their ability to support contract evolution and control over the target logic contract: Evolvable Proxy Contracts (EPCs) and Forwarder Proxy Contracts (FPCs) (Jorge Izquierdo 2018; Salehi et al. 2022; Bodell III et al. 2023). Each serves different architectural and operational purposes in Ethereum smart contract systems.

2.15.5.1 Upgradeable Proxy Contracts (UPCs)

An Upgradeable Proxy Contract (UPC) is designed to support future changes to the logic contract without requiring changes to the contract address (i.e., the proxy contract address) through which users interact. The key defining feature of this proxy type is its ability to dynamically switch the logic contract to which it delegates calls. This is typically achieved by maintaining a modifiable state variable, often called the implementation variable, within the proxy's storage, which holds the address of the current logic contract (Listing 2.3, line 5).

To evolve the application's behavior, developers or designated administrators must first deploy a new version of the logic (implementation) contract that includes the desired modifications, such as bug fixes, technical debt repayment, new features, performance improvements, or security patches. Once deployed on-chain, the address of this new logic contract becomes the target for delegation.

The upgrade process is initiated by invoking a dedicated function typically defined in the proxy contract (e.g., `upgradeTo(address newImplementation)`), which replaces the current value of the implementation variable with the address of the new logic contract (Listing 2.3, lines 13–17). This action effectively updates the proxy's delegation path.

As a result, all subsequent external calls to the proxy are automatically forwarded to the new logic contract via `delegatecall`, while continuing to read from and write to the same persistent storage maintained by the proxy. Because the new logic contract's code is executed in the proxy's context, this upgrade seamlessly alters the application's behavior without modifying any existing state, including user balances, permissions, or data structures. From the caller's perspective, the application interface and contract address remain unchanged, but internally, it now operates under a new implementation.

This contract evolution mechanism enables Ethereum-based applications to evolve over time, supporting upgrades and maintenance in-place without requiring data migration, user reconfiguration, or redeployment of the proxy contract.

2.15.5.2 Forwarder Proxy Contracts (FPCs)

A Forwarder Proxy Contract is designed to permanently delegate calls to a fixed, unchanging logic contract. Once deployed, the address of the logic contract is hard-coded or immutably set in the proxy's constructor (Listing 2.3, lines 8–11), and cannot be modified post-deployment. These proxies are lightweight and deterministic, often used to reduce gas costs and deploy many instances of the same logic in a scalable way.

2.15.6 UPC Patterns

This section presents the well-established upgradeable proxy contract patterns and proposals that are built on the proxy design pattern. Meisami and III (2023) provide a comprehensive list of recognized proxy patterns, from which we selected eight

that are widely adopted to support contract evolution. These patterns serve as the foundation for our analysis in Chapter 6. For brevity, we refer to these as *UPC Patterns* throughout the remainder of this thesis.

2.15.6.1 Inherited Storage UPC Pattern

This pattern is designed to facilitate seamless smart contract upgrades while ensuring consistent storage across different versions, thereby maximizing both flexibility and reliability in contract evolution (OpenZeppelin 2018a). Figure 2.4 presents a simplified class diagram illustrating this pattern. The `UpgradeabilityStorage` contract holds the address of the implementation contract, ensuring that every new implementation inherits this setup via the `Upgradeable` contract. This inheritance chain preserves the storage layout defined in earlier versions and prevents accidental overwrites of the implementation contract address by newer implementations. The `Proxy` contract serves as the front-facing interface for all interactions and is responsible for delegating calls to specific implementation contracts. The `UpgradeabilityProxy` inherits from both `Proxy` and `UpgradeabilityStorage`, enabling it to manage which version of the implementation is currently active and allowing seamless contract evolution.

2.15.6.2 Eternal Storage UPC Pattern

This pattern is designed to support robust and flexible smart contract evolution while preserving a consistent and immutable storage structure (OpenZeppelin 2018a). Figure 2.5 presents a simplified class diagram of this pattern. The `UpgradeabilityStorage`

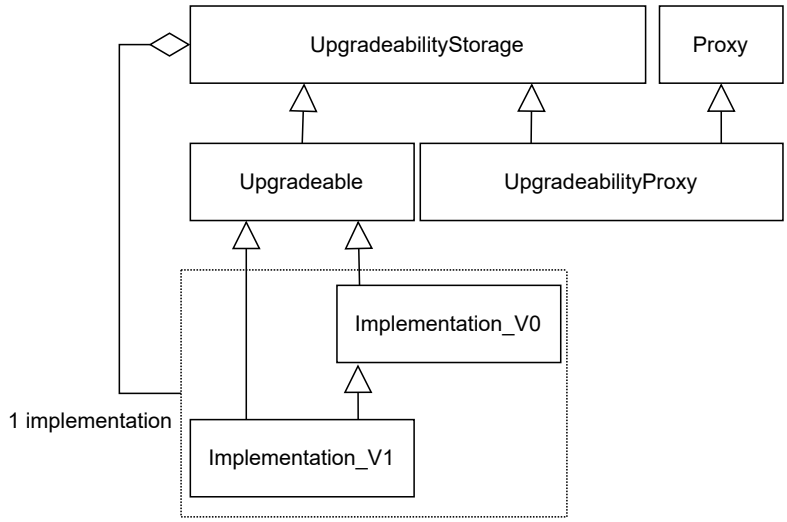


Figure 2.4: Key components of the Inherited Storage UPC Pattern.

contract holds essential upgrade-related data (e.g., the implementation contract address). The `Proxy` contract routes interactions to the appropriate implementation version via `delegatecall`. The `UpgradeabilityProxy` combines the functionalities of both `UpgradeabilityStorage` and `Proxy` to facilitate seamless transitions between implementations.

A key component of this pattern is the `EternalStorage` contract, which defines a universal and immutable storage schema composed of mappings for each variable type. This storage schema is inherited by all subsequent implementation contracts and the `EternalStorageProxy` contract. Such direct inheritance ensures that all upgrades and behavioral modifications remain fully compatible with the original storage layout defined by `EternalStorage`, thereby preventing inconsistencies in data handling across versions. To maintain data consistency, each new implementation contract must conform to the storage schema established by its predecessor.

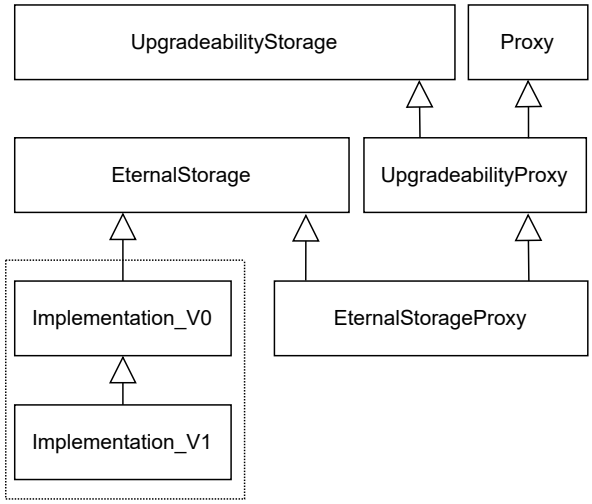


Figure 2.5: Key components of the Eternal Storage UPC Pattern.

2.15.6.3 Registry UPC Pattern

This pattern offers a centralized, manageable, and secure solution for tracking and updating different versions of contract implementations (Peh 2018). Figure 2.6 illustrates a simplified class diagram of this pattern. The `IRegistry` interface defines the structure and expected functionality for a `Registry` contract. The `Registry` implements this interface and is deployed at a separate Ethereum address. It maintains a registry of version identifiers mapped to their corresponding implementation contract addresses, allowing the owner to register new versions or query the implementation address associated with a specific version.

The `Proxy` contract acts as a forwarder, delegating calls to the current implementation. The `UpgradeabilityProxy` is a specific instance of the `Proxy` that queries the `Registry` before delegating each call, ensuring that it always routes to the latest implementation address associated with the current version. This ensures that

the `UpgradeabilityProxy` always uses the latest version registered in the Registry without requiring manual updates, enabling more dynamic contract evolution.

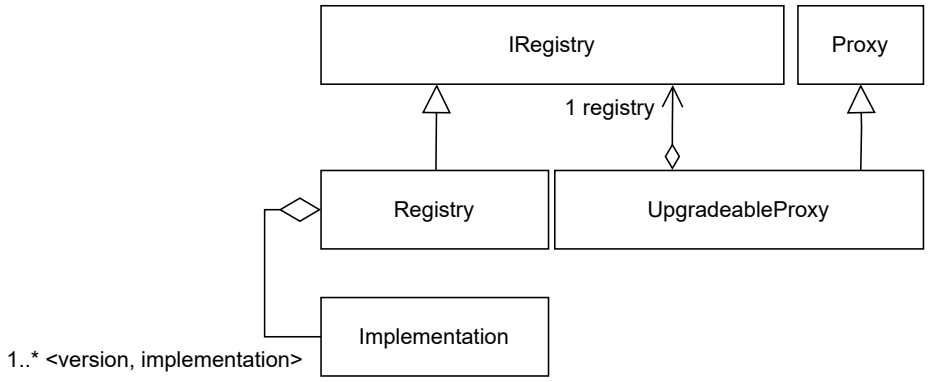


Figure 2.6: Key components of the Registry UPC Pattern.

2.15.6.4 Beacon UPC Pattern

(Dharma 2019). This pattern is designed to facilitate the dynamic updating of implementation contracts, offering an efficient and centralized mechanism for managing upgrades across multiple proxies. Figure 2.7 illustrates a simplified class diagram of this pattern. The `IBeacon` interface defines the minimal requirements for beacon contracts, primarily, the ability to return the current implementation address.

The `UpgradeableBeacon` is a concrete implementation of the `IBeacon` interface that supports dynamic updates of the implementation address through an upgrade function. The `Proxy` contract acts as a stable intermediary that delegates all function calls to the target implementation address. The `BeaconProxy` extends this `Proxy` functionality by integrating directly with a beacon contract (conforming to the `IBeacon` interface) to retrieve the latest implementation address before each

call. This design ensures that the proxy always delegates calls to the most recent implementation, thereby achieving dynamic upgradeability while maintaining operational consistency.

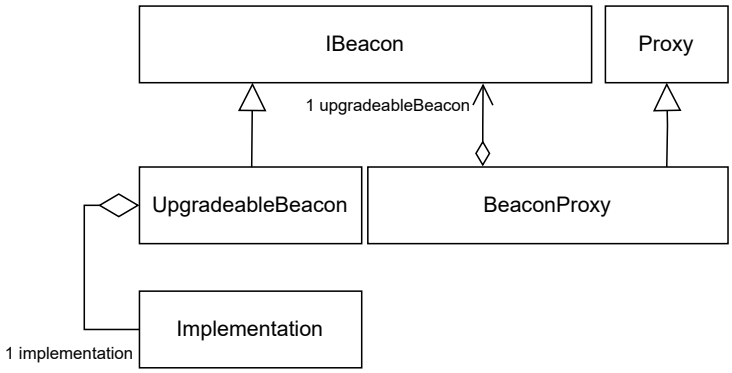


Figure 2.7: Key components of the Beacon UPC Pattern.

2.15.6.5 ERC-1967 UPC Pattern (Unstructured Storage)

(Santiago Palladino 2019a). This pattern incorporates the concept of *Unstructured Storage*, introduced by OpenZeppelin, in which proxy-specific data, such as the implementation contract address, is stored in designated fixed storage slots that are not allocated by the compiler. These slots are computed using a hashing mechanism. Specifically, a string identifier (e.g., `"eip1967.proxy.implementation"`) is hashed using the `keccak256` hashing function. This technique ensures that critical proxy-related values do not interfere with the storage layout of the implementation contracts, thereby avoiding accidental overwrites.

Figure 2.8 presents a simplified class diagram illustrating this pattern. The `Proxy` serves as the central interface, delegating interactions to the appropriate implementation contracts via `delegatecall`. The `ERC1967Upgrade` contract provides a suite

of functionalities for managing and executing smart contract upgrades. This abstract contract utilizes explicitly defined storage slots to control various aspects of the proxy, including the current implementation address, the admin address, and optionally a beacon address.

The `ERC1967Proxy` contract inherits from both `Proxy` and `ERC1967Upgrade`, integrating upgrade functionality into a secure and modular structure accessible through public interfaces. This standard has been widely adopted across other UPC patterns and is regarded as a best practice for architecting proxy storage layouts.

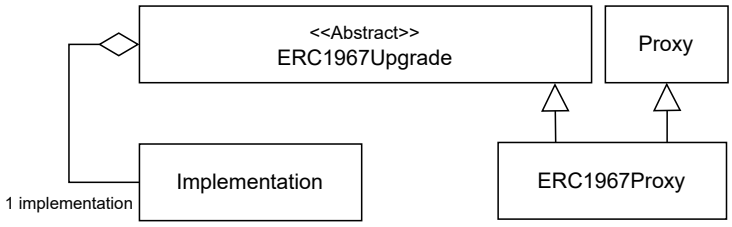


Figure 2.8: Key components of the ERC-1967 UPC Pattern.

2.15.6.6 ERC-1822 Universal UPC Pattern (UUPS)

(Gabriel Barros 2019a). This pattern offers a flexible and cost-efficient approach to upgrading smart contracts while preserving contract state and reducing gas consumption. Figure 2.9 presents a simplified class diagram illustrating this pattern. The `ERC1967Utils` library defines and manages storage slots in accordance with the ERC-1967 standard (also known as unstructured storage) for upgradeable proxies, specifically focusing on organizing the storage structure to safely and transparently manage updates to the implementation address.

The `Proxiable` contract uses the specific implementation storage slot defined by `ERC1967Utils` to securely initiate upgrades. Similarly, the `UUPSProxy` leverages

the same utility library to read from the designated implementation storage slot. To enable secure upgrade operations, each implementation contract must conform to the `Proxiable` interface. This consistency guarantees that all implementation contracts write to the same storage slot that the `UUPSProxy` reads from.

To perform an upgrade, the contract owner invokes the `upgrade` function via the `UUPSProxy`, which delegates the call to the corresponding function in the implementation contract. Since `delegatecall` executes in the context of the `UUPSProxy`, any update to the implementation address is reflected directly in the proxy's storage, thereby completing the contract evolution securely and efficiently.

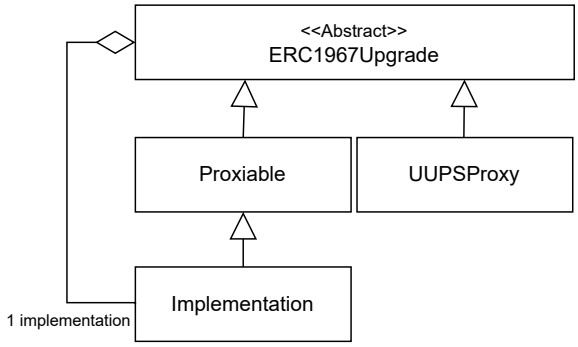


Figure 2.9: Key components of the ERC-1822 Universal UPC Pattern (UUPS).

2.15.6.7 ERC-2535 Diamonds UPC Pattern

This pattern provides a sophisticated and modular framework for Ethereum smart contracts, enabling scalable and modular contract evolution (Mudge 2020b). The standard introduces a `DiamondProxy` contract that delegates functionality to multiple `Facet` contracts (i.e., implementation contracts). Each `Facet`, deployed independently, can contribute specific functions to the `DiamondProxy`, enhancing the

modular design of the system. Figure 2.10 presents a simplified class diagram illustrating this pattern.

The `DiamondProxy` serves as the central contract, delegating calls to specific functions implemented in various `Facets`. The `LibDiamond` library facilitates the addition, replacement, and removal of functions, manages diamond storage, maps function selectors to facets, and enforces access control. The `IDiamondCut` interface defines the structure and rules for modifying facets by adding, replacing, or removing their functions. The `IDiamondLoupe` interface provides introspection capabilities to query available facets and their respective function selectors within a diamond.

Two essential facet contracts implement these interfaces: `DiamondCutFacet` (for performing upgrades) and `DiamondLoupeFacet` (for querying facet composition). These facets interact with the `DiamondProxy` through `delegatecall`, allowing the proxy to execute logic defined in the facets while maintaining a single storage context. Notably, facets do not store data within their own contract storage; instead, they access and manipulate state variables stored in the `DiamondProxy`.

Similar to the ERC-1822 UPC Pattern, upgrades are performed by delegating control from the `DiamondProxy` to the `DiamondCutFacet`'s `diamondCut` function. Because `delegatecall` operates within the proxy's context, changes to the function selector mappings are reflected in the proxy's storage, thereby enabling seamless and dynamic upgrades.

2.15.6.8 Transparent UPC Pattern

(Palladino 2021). Upgradeable proxies, which delegate function calls to an implementation contract, can encounter function signature clashes due to Solidity's

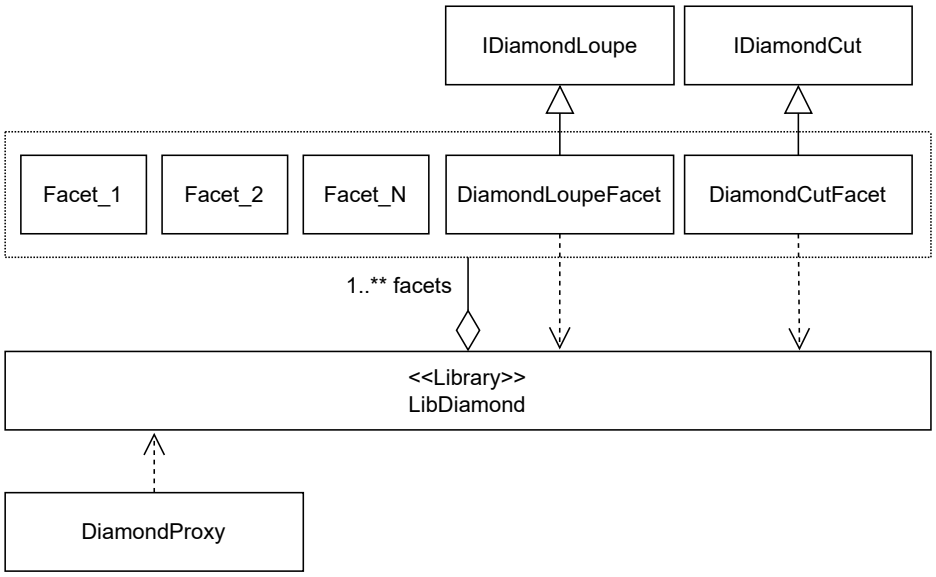


Figure 2.10: Key components of the ERC-2535: Diamonds UPC Pattern.

use of 4-byte function identifiers. This encoding may cause different functions from separate contracts (e.g., the proxy and its implementation contracts) to share the same identifier, potentially resulting in the execution of incorrect functions when called through the proxy.

The Transparent UPC Pattern addresses this issue by differentiating behavior based on the identity of the caller. If the caller is the admin (i.e., the account authorized to manage contract evolution), the proxy processes the call directly. For all other callers, the proxy delegates the function call to the implementation (logic) contract. This design prevents administrators from inadvertently invoking application logic when attempting to manage the proxy, thereby enhancing operational control and security.

Figure 2.11 presents a simplified class diagram for this pattern. As shown, it reuses components from the ERC-1967 UPC Pattern and introduces the `Transparent`

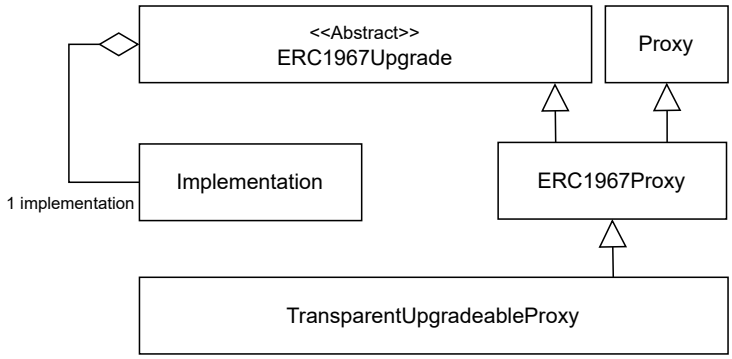


Figure 2.11: Key components of the Transparent UPC Pattern.

`UpgradeableProxy` component, which evaluates the caller's identity and redirects calls accordingly.

CHAPTER 3

Literature Survey

In recent years, the increasing complexity, criticality, and pervasiveness of software systems have heightened the importance of understanding and managing technical debt (TD). Technical debt refers to the long-term costs and risks associated with taking design or implementation shortcuts to accelerate development. A growing body of research has focused on a specific form of TD known as Self-Admitted Technical Debt (SATD), where developers explicitly acknowledge suboptimal decisions or temporary workarounds through comments in the source code. This chapter presents a comprehensive literature review covering the evolution of SATD detection techniques, the characteristics of SATD across various software domains, and the development of supporting tools and taxonomies. We also

examine research on the evolution and maintainability of smart contracts, particularly through different upgrade mechanisms such as the proxy pattern, highlighting key classification efforts and detection approaches. Finally, we position our contributions in relation to this body of work, emphasizing how our large-scale empirical analyses and tooling address critical gaps in understanding and managing smart contract evolution on Ethereum.

3.1 Self-Admitted Technical Debt (SATD)

Cunningham (1992) introduced the original technical debt (TD) metaphor in an experience report based on the development of WyCASH+, an object-oriented financial portfolio management system built in Smalltalk. This report coined the now widely used analogy between shipping immature code and incurring debt: *Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. The danger occurs when the debt is not repaid.* This work highlighted the concept of intentional technical debt, where developers consciously take shortcuts to achieve time-to-market advantages, with the expectation of future refactoring. Cunningham also emphasized incremental development and iterative design repair, viewing architectural evolution as a process of accumulating and repaying debt.

The concept of TD was later extended by Potdar and Shihab (2014) to include Self-Admitted Technical Debt (SATD), a specific type of TD explicitly acknowledged

by developers through source code comments that highlight workarounds, incomplete implementations, or known issues. Over the past decade, approaches to detecting SATD in software development have evolved substantially, moving from simple natural language techniques to sophisticated deep learning models. Sutoyo and Capiluppi (2024) conducted a comprehensive systematic literature review of SATD detection approaches from 2014 to 2024, categorizing the evolution into four primary phases: traditional Natural Language Processing (NLP), Machine Learning (ML), Deep Learning (DL), and Transformers-based models.

Early SATD detection approaches primarily relied on traditional NLP techniques, such as part-of-speech tagging, regular expressions, and textual pattern matching, which focused on manually engineered features to identify SATD in code comments (e.g., keywords like `TODO` or `FIXME`). These methods provided foundational insights but struggled with scalability and recall (Potdar and Shihab 2014; S. Maldonado and Shihab 2015; de Freitas Farias et al. 2015; Bavota and Russo 2016; Guo et al. 2019; Freitas Farias et al. 2020).

As the complexity of software systems increased, researchers began employing machine learning classifiers, such as Naïve Bayes, Support Vector Machines (SVM), k-Nearest Neighbors (kNN), Random Forest (RF), and XGBoost, to automate SATD detection. These methods improved performance by learning from labeled data and reduced the need for handcrafted features (Maldonado et al. 2017a; Liu et al. 2018; Huang et al. 2017; Flisar and Podgorelec 2019; Wattanakriengkrai et al. 2019; Maipradit et al. 2020; Yu et al. 2020; Sala et al. 2021; Chen et al. 2022d).

The introduction of deep learning techniques marked another significant leap. Models like Convolutional Neural Networks (CNN), Long Short-Term Memory networks (LSTM), and Bidirectional LSTM (BiLSTM) enabled the automatic learning of complex textual patterns in software artifacts, improving both precision and recall in SATD detection (Ren et al. 2019; Santos et al. 2020; Wang et al. 2021; Sridharan et al. 2023).

Most recently, Transformers-based models (such as BERT, RoBERTa, and GPT) have demonstrated state-of-the-art performance in both SATD identification and SATD categorization tasks. These models excel in capturing contextual and semantic nuances in software text, leading to robust and generalizable SATD detection across diverse projects (Prenner and Robbes 2021; Qu et al. 2022b; Sheikhaei et al. 2024).

In the following subsections, we first review key studies on SATD across four domains: open-source software, machine learning systems, enterprise software, and blockchain infrastructure. We then explain the rationale for our choice of SATD detector, which will be employed in Chapter 3.1.5 for our study.

3.1.1 Open-Source Software

Potdar and Shihab (2014) introduced SATD as code comments in which developers explicitly acknowledge sub-optimal code implementations. The study aimed to investigate the prevalence and patterns of SATD in open-source projects (OSS). The authors analyzed four large OSS projects (Eclipse, Chromium OS, Apache HTTP Server, ArgoUML), extracting approximately 101k source code comments. Through

manual inspection using the srcML toolkit, they identified 62 recurring textual patterns (e.g., `TODO`, `FIXME`, `hack`) that indicate SATD. They catalogued these patterns as a basis for future detectors. SATD comments were common, up to 31% of files contained self-admitted debt in some projects, suggesting developers often knowingly inject technical debt and leave remarks. The study highlighted the potential maintenance risk of SATD and called for techniques to detect and manage it.

de Freitas Farias et al. (2015) introduced CVM-TD (Contextualized Vocabulary Model for Identifying Technical Debt on Code Comments), an early approach that explored the use of natural language structures to improve the identification of SATD. The authors propose CVM-TD as a structured vocabulary model, combining software engineering-related nouns, adjectives, verbs, adverbs, and commonly used tags (e.g., `TODO`, `FIXME`, `NOTE`) to systematically compose patterns indicative of TD. Their exploratory study used two large, mature open-source Java systems, jEdit and Lucene, totaling over 6,000 useful comments, extracted and processed using their custom eXcomment tool. The identification pipeline combined manual feature engineering (POS tagging, ontology-based noun mapping) with keyword-based heuristics and contextual pattern composition. Results show that between 12.87% and 17.31% of filtered comments in the two projects were assessed by expert annotators as potential TD indicators, with code and design debt being the most prominent.

S. Maldonado and Shihab (2015) extended SATD research by focusing on classification of SATD types. The authors manually inspected code comments from five open-source projects, introducing heuristics (e.g. filtering out license headers,

parsing with JDeodorant) to isolate true SATD comments. They proposed a taxonomy of five SATD categories including Design debt, Defect (bug) debt, Documentation debt, Requirement debt, and Test debt. They then quantified each category's frequency, finding design-related debt to be most prevalent, whereas defect, doc, and test debts were relatively rarer. The study provided a first dataset of labeled SATD instances (33,093 comments) and observed that design (ranging approximately 55–95% in projects) and requirement debts (ranging approximately 5-45% in projects) together often made up the majority of SATD while other types were less than 10%.

Bavota and Russo (2016) present a large-scale empirical study on SATD, extending the earlier work of Potdar and Shihab (2014). The study investigates the diffusion, evolution, and impact of SATD across a broad set of software systems. The dataset includes 159 Java open-source projects from the Apache and Eclipse ecosystems, encompassing 658K commits and over 2 billion mined comments. SATD identification is performed using a pattern-based heuristic with regular expressions using the 62 SATD patterns established by Potdar and Shihab, followed by manual classification of a statistically significant sample of 366 SATD comments to refine debt categories. The study adopts an SATD taxonomy derived from Alves et al. (2014), covering code, defect, requirement, design, test, and documentation debt. Results show that SATD is widespread (averaging 51 instances per system, or 0.3% of comments), with code debt (30%), defect debt (20%), and requirement debt (20%) being the most frequent categories. The number of SATD instances tends to increase over time, and removed instances persist for an average of 1,087 commits. Additionally, 63% of SATD removals are performed by the original author.

The study finds no significant correlation between internal code quality metrics and SATD presence, suggesting that SATD introduction is often independent of code complexity or readability.

Maldonado et al. (2017a) introduced the first automated SATD detection approach using machine learning on textual features. The authors leveraged the labeled dataset of SATD comments from their previous study (S. Maldonado and Shihab 2015) and extended it to ten open-source projects, comprising approximately 62,275 comments, of which 4,071 are labeled as SATD. Then, they built a classifier (maximum entropy model) with features from comment text (e.g. n-grams, semantic cues) to identify design and requirement SATD. To address the challenge of vocabulary diversity and project-specific terms in comments, the approach included cross-project training and feature selection. Their NLP-based detector significantly outperformed the earlier pattern-based method of Potdar and Shihab (2014), achieving an average F1-score improvement of 2.3x and 6x across design and requirement debt, respectively.

Huang et al. (2017) improved SATD detection performance and tool support via an ensemble text-mining approach. The authors used a feature selection technique to pick useful words from comment text and trained multiple classifiers on different projects, then combined them into a composite model. They evaluated on eight large Apache projects (approximately 212k comments). By leveraging diverse project data, their approach achieved robust results on new projects. On each target project, the ensemble detector outperformed both the state-of-the-art (Maldonado et al. 2017a) and baseline learners in F1-score. The approach attained F1-score ranging 0.518–0.841, significantly improving recall of SATD compared to

prior methods. For example, it achieved a 499% relative F1-score improvement over the simple pattern-based approach. Later on, Liu et al. (2018) released this method as an Eclipse plugin, which highlights and lists detected debt comments within the IDE.

Flisar and Podgorelec (2019) propose an enhanced method for SATD identification that leverages word embeddings to improve feature selection in SATD classifiers. The study aims to address limitations of prior Bag-of-Words and pattern-based models. The study adopts five common SATD types (design, defect, documentation, requirement, test debt), but focuses on binary classification (SATD vs. non-SATD). The dataset comprises 62,275 labeled comments from ten open-source Java projects, sourced from the Maldonado et al. (2017a)'s dataset. The proposed method is a hybrid approach that combines traditional feature selection techniques (CHI, IG, MI) with word2vec-based feature enhancement. A domain-specific word embedding model was trained on 1.2 million unlabeled source code comments from 360 Java GitHub projects (110K unique tokens), enabling the selection of semantically similar terms to enrich feature vectors. The classifiers used include Naive Bayes, SVM, and Maximum Entropy, with TF-IDF weighting. Results show that the enhanced feature selection with word embeddings (FS+WE) significantly outperforms baseline feature selection and existing similarity-based methods (SE-WordSim, WS4J), achieving up to 82% F1-macro with SVM+MI and 30% feature selection. The study demonstrates that embedding-based feature enhancement improves SATD identification by capturing semantic relations among comment terms, particularly in diverse open-source projects.

Wattanakriengkrai et al. (2019) propose a multi-class classification approach to automatically detect and categorize SATD. The study focuses on the two most common SATD types: design debt and requirement debt. The identification method employs N-gram Inverse Document Frequency (IDF) feature extraction, combined with instance hardness under-sampling to address class imbalance. Classification is performed using a Random Forest model trained on the dataset from Maldonado et al. (2017a). The features used are purely lexical, based on N-gram IDF representations. The proposed approach outperforms traditional baselines, including Bag-of-Words, TF-IDF, and SVM models, achieving an F1-score of up to 0.874 on the ArgoUML project. Additionally, it improves requirement SATD classification performance by 33% compared to the benchmark model by Maldonado et al. (2017b).

Ren et al. (2019) advanced SATD detection using deep learning and investigated model interpretability. The authors designed a Convolutional Neural Network (CNN) classifier that learns representations of code comments to distinguish SATD. The CNN's convolution kernels capture local phrases (e.g., "need to refactor" or "temporary fix") that are strong SATD signals. They trained on 62,566 comments from ten open-source projects, using the labeled dataset by Maldonado et al. (2017a), and evaluated the model on new projects to assess generalizability. Importantly, to address explainability, they devised a method to trace the CNN's predictions back to specific phrases in the comment text, essentially highlighting which tokens (e.g., "ugly hack") triggered the classifier. The CNN approach achieved superior accuracy and adaptability in cross-project tests, outperforming traditional text mining baselines on precision/recall. The user study confirmed that the phrases

highlighted by the model (via its built-in back-tracing mechanism) aligned with human intuition for why a comment indicates debt.

Freitas Farias et al. (2020) propose an improved pattern-based approach for supporting the identification and classification of SATD. The study refines the SATD taxonomy based on Alves et al. (2014) and Rios et al. (2018), covering nine types: design, code, architecture, test, documentation, defect, requirement, people, and build debt. The dataset consists of three large open-source Java projects (ArgoUML, jEdit, and Lucene), analyzed using a contextualized SATD vocabulary comprising 349 patterns, developed from their prior work (de Freitas Farias et al. 2015). Identification is performed through pattern-based comment mining with the eXcomment tool, which incorporates a ranked vocabulary where patterns are weighted by decisiveness and mapped to debt types. Results indicate that the improved vocabulary enhances both SATD identification precision and debt type classification, with the most frequently detected types being code, design, defect, documentation, and requirement debt.

Maipradit et al. (2020) present an automated approach for detecting a specific subtype of SATD, termed On-hold SATD, aimed at improving SATD identification and management. On-hold SATD refers to comments where the resolution of technical debt is explicitly dependent on an external event (e.g., issue closure). The study adopts the general five-type SATD taxonomy from S. Maldonado and Shihab (2015) but focuses exclusively on On-hold SATD with issue references. The dataset comprises 1,530 issue-referencing comments mined from ten large open-source Java projects. Identification is performed via a hybrid approach: initial regular expression matching extracts issue-referencing comments, followed by manual

annotation (On-hold SATD vs. cross-reference), achieving inter-annotator agreement (Cohen's kappa = 0.748). A machine learning classifier (auto-sklearn with n-gram IDF features) is then used to distinguish On-hold SATD. Evaluation via 10-fold cross-validation yields strong results (Precision = 0.79, Recall = 0.70, F1-score = 0.73, AUC = 0.97). The study also analyzes SATD evolution, reporting a median lifespan of 42 days for On-hold SATD, with 13% persisting for over a year even after the external condition has been fulfilled.

Yu et al. (2020) propose Jitterbug, a two-step framework for identifying SATD, aiming to improve both identification. The authors distinguish two types of SATD: "easy-to-find" (e.g., containing `TODO`, `FIXME`) and "hard-to-find" SATDs. Their approach first employs a pattern recognition technique to automatically identify the "easy-to-find" SATDs with close to 100% precision, followed by a human-in-the-loop machine learning component to guide expert review of the remaining "hard-to-find" SATDs. The study uses the well-established dataset by Maldonado et al. (2017a). The focus is on binary SATD detection (SATD vs. non-SATD), not SATD type classification. The tool Jitterbug is open source and standalone. Evaluation shows that Jitterbug outperforms prior baselines such as CNN (Ren et al. 2019), and MAT (Guo et al. 2021), achieving near-100% precision for easy SATDs and significantly reducing human effort (saving up to 23 hours per project).

Wang et al. (2021) propose a novel deep learning-based approach to improve both identification and explainability of SATD detection. The study adopts the common five-type SATD taxonomy introduced by S. Maldonado and Shihab (2015), but focuses on binary classification (SATD vs. non-SATD). The dataset consists of 445,365 comments from 20 open-source projects, including the 10 Java projects

from Maldonado et al. (2017a)'s benchmark dataset and 10 additional real-world projects from GitHub spanning Java, Python, JavaScript, and multi-language systems. The identification method is HATD (Hybrid Attention-based method), a deep learning architecture combining ELMo embeddings, Bi-LSTM, and both single-head and multi-head attention mechanisms. Features are dynamic contextual embeddings (ELMo), augmented with attention-based sequence modeling. HATD achieves an average F1-score of 84.22% within-project and 82.99% cross-project, consistently outperforming Ren et al. (2019)'s CNN-based baseline by 9.14% (within-project) and 15.61% (cross-project). Additionally, the study demonstrates HATD's ability to generate interpretable explanations via attention visualizations and extracts new SATD patterns beyond Potdar's 62 baseline patterns.

Santos et al. (2020) propose an approach for improving identification of SATD in source code comments using a combination of Long Short-Term Memory (LSTM) neural networks and Word2vec word embeddings. The study adopts the five common SATD types defined by Alves et al. (2014), but focuses specifically on detecting design debt and requirement debt. The dataset consists of 62,275 manually labeled comments (4,071 SATD instances), extracted from ten open-source Java projects compiled by Maldonado et al. (2017a). The identification method is a deep learning model combining Word2vec embeddings trained on 1.2M Java comments from 360 GitHub repositories, with an LSTM neural network trained on the labeled SATD dataset. Baselines for comparison include Auto-sklearn (Wattanakriengkrai et al. 2019) and Maximum Entropy classifiers (Maldonado et al. 2017b). Evaluation uses leave-one-out cross-project validation across the 10 projects. Results show that for

design SATD, LSTM with Word2vec improves recall and F1-score over both baselines (average F1-score = 0.67%), with statistically significant gains confirmed via t-tests. For requirement SATD, Auto-sklearn achieves higher precision, but LSTM with Word2vec obtains competitive F1-scores and improved recall, important for recall-sensitive SATD detection tasks.

Guo et al. (2021) conduct a comprehensive empirical study to critically assess the state of progress in identification of SATD by systematically comparing existing approaches with a simple task tag-based heuristic. The study focuses on binary classification (SATD vs. non-SATD). The datasets consist of two large collections: (1) Dataset-M, the 10 open-source Java projects from Maldonado et al. (2017a), and (2) Dataset-G, a newly collected dataset of 10 additional Java projects including Dubbo, Gradle, Groovy, Hive, Maven, Poi, SpringFramework, Storm, Tomcat, and Zookeeper, comprising 81,260 comments with 3K SATD instances. The identification method centers on MAT (Matching task Annotation Tags), a simple heuristic that fuzzily matches common task tags (`TODO`, `FIXME`, `XXX`, `hack`) in comments. MAT is compared against five representative SATD identification baselines: Pattern matching (Potdar and Shihab 2014), NLP-based Maximum Entropy (S. Maldonado and Shihab 2015), TM (Text Mining) (Huang et al. 2017), CNN-based (Ren et al. 2019), and Jitterbug (Yu et al. 2020). Features for MAT are purely lexical patterns (tags), with preprocessing and fuzzy matching. Evaluation covers Many-to-One and One-to-One cross-project prediction scenarios, using Precision, Recall, F1, and effort-aware metrics (ER, RI). Results reveal that MAT achieves similar or even superior performance to state-of-the-art models in projects where task tags are frequently used, with highly overlapping correctly classified instances. Notably,

many supervised models (including CNN) misclassify SATDs marked with task tags, which can be corrected by combining MAT. The study highlights that current SATD identification progress may be overestimated in the literature, advocating that MAT should serve as a baseline in future SATD research to enable more realistic assessments.

Prenner and Robbes (2021) investigated whether pre-training, combined with techniques like data augmentation, self-training, soft labels, intermediate-task fine-tuning, and active learning, can improve performance on small datasets; the focus spans identification and classification tasks, including SATD detection. The study uses 13 SE datasets (natural language, code, comments), including the Maldonado et al. (2017a)'s SATD dataset. SATD identification is performed using Transformer-based models (BERT, RoBERTa, domain-specific variants), compared against baselines like CNNs and SVMs, with features derived via contextual embeddings. The study demonstrated that pre-trained Transformer models achieved state-of-the-art results for SATD detection (macro F1-score approximately 81%), surpassing prior CNN and traditional baselines, and highlighted that while domain-specific pre-training did not yield further gains, additional techniques such as self-training could enhance recall when desired.

Sala et al. (2021) propose DebtHunter, a machine learning-based approach for the identification and classification of SATD. The study adopts the five common SATD types defined by Alves et al.Alves et al. 2014: design, defect, documentation, requirement, and test debt. The dataset consists of 259,229 labeled comments (62,566 SATD instances) from ten open-source Java projects, originally curated

and labeled by Maldonado et al. (2017a). DebtHunter employs a two-step classification pipeline: first, a binary classification of SATD versus non-SATD; second, a multi-class classification of SATD instances into the respective debt types. The classifier used is Sequential Minimal Optimization (SMO), optimized through grid search. The study also investigates the effects of feature selection (Information Gain) and data sampling techniques (SMOTE for oversampling and SpreadSubsample for undersampling) on classification performance. DebtHunter demonstrates superior performance compared to five baseline approaches, including pattern-based detection (Potdar and Shihab 2014), binary classification (Liu et al. 2018), and the MAT (Guo et al. 2019). In binary classification, DebtHunter achieves high performance (Precision = 0.972%, Recall = 0.967%, F1-score = 0.965%), while in multi-class classification, the best configurations yield notable improvements in the detection of requirement and defect debt compared to prior methods.

Chen et al. (2022d) propose a multiclass classification approach for the identification and classification of SATD. The study adopts a simplified taxonomy focused on three types of SATD: defect debt, design debt, and implementation debt, building on prior taxonomies from Alves et al. (2014) and S. Maldonado and Shihab (2015). The authors used Maldonado et al. (2017a)'s SATD dataset for training and evaluation. The proposed identification pipeline includes four main stages: (1) preprocessing (tokenization, stopword removal, lemmatization), (2) data augmentation using Easy Data Augmentation (EDA) to address class imbalance, (3) feature selection with Chi-Square statistics, and (4) classification using XGBoost with CountVectorizer features. The approach is evaluated against four baselines: (1) NLP-based Maximum Entropy classifier (Maldonado et al. 2017a), (2) Naive Bayes

Multinomial (Huang et al. 2017), (3) a CNN-based method (Ren et al. 2019), and (4) a BiLSTM-based method (Yu et al. 2021). Evaluation is conducted using leave-one-out cross-project validation with Macro Precision, Macro Recall, and Macro F1-score as metrics. The proposed XGBoost model achieves an average Macro F1-score of 55.77%, Macro Precision of 56.66%, and Macro Recall of 59.07% across the ten projects, outperforming the NLP-based baseline by 3–5 percentage points.

Sridharan et al.Sridharan et al. (2023) addressed limitations of prior SATD datasets by constructing PENTACET, a dataset of 23 million contextual code comments (including more than 250,000 SATD comments) mined from 9,096 open-source Java projects (over 400 million LOC). The objective is to support more sophisticated SATD identification, especially for "hard-to-find" SATD beyond simple tags (e.g., 'TODO', 'FIXME'), through context-aware analysis. The dataset is built using open-source projects from GitHub, filtered by quality and activity criteria, and includes detailed metadata. SATD comments were identified via a semi-automated process combining Sense2Vec-based feature expansion, manual heuristics, and validation with SoCCMiner and NLoN tools, achieving substantial inter-rater agreement (Cohen's Kappa = 0.75). While the study does not propose a new taxonomy, it extends feature-based SATD classification by introducing 655 "hard-to-find" features.

More recently, Sheikhaei et al. (2024) evaluated the effectiveness of Large Language Models (LLMs) for SATD identification and classification. The study systematically explored fine-tuned and in-context learning (ICL) variants of LLMs, specifically Flan-T5, BERT, and CodeBERT, in comparison to prior state-of-the-art models (e.g., CNN). The evaluation used two public datasets: (1) Maldonado et al. (2017a)'s dataset and (2) OBrien et al. (2022)'s dataset, the latter augmented with

contextual features (file path, method signature/body). Models were trained using supervised fine-tuning and ICL approaches, incorporating LLM embeddings and architectural modifications (e.g., Flan-T5 with a classification head). Results show that fine-tuned LLMs, particularly Flan-T5-XL, outperform baselines, achieving a 4.4–7.2% F1-score improvement in SATD identification and up to 66.8% classification accuracy with contextual features. The study highlights that larger LLMs effectively leverage contextual code information to enhance classification, while ICL remains competitive but underperforms compared to fine-tuned models.

3.1.2 Machine Learning Systems

OBrien et al. (2022) conducted a large-scale empirical study of SATD in machine learning (ML) projects, introducing a detailed taxonomy of ML-related technical debt. They mined 68,820 SATD comments from the full revision history of 2,641 popular open-source ML repositories on GitHub. The analysis examined when and where SATD comments appear across ML pipeline stages (from data preprocessing and model training to validation and deployment) and compared patterns between ML applications (end-user ML-powered software) and ML tools/frameworks (e.g., TensorFlow).

The study produced a taxonomy of 23 SATD types. While some categories overlapped with classic debt (e.g., code and documentation debt), others were unique to ML, such as data debt, model debt, and pipeline debt. The distribution of SATD types varied by pipeline stage: data preprocessing and model training contained higher concentrations of debt, whereas validation and deployment stages exhibited fewer debt comments. The study also found that much SATD in ML code persists

for long durations, indicating postponed debt repayment. Additionally, ML applications tended to accumulate more pipeline-specific debt (related to data and experimentation), whereas ML frameworks contained more engineering debt (design and test-related), reflecting differing development priorities.

Bhatia et al. (2024a) conducted a comparative study of SATD in machine learning (ML) versus traditional software, with insights into the evolution and longevity of ML-related debts. They analyzed two cohorts: 318 ML repositories and an equal set of 318 non-ML repositories. Using automated classifiers and manual qualitative analysis on sampled SATD comments, the authors examined the nature of SATD in ML contexts and performed survival analysis to assess its persistence.

The study revealed that: (i) ML projects exhibit substantially more SATD , the median percentage of files with SATD is about twice that of non-ML projects, likely due to the rapid experimentation inherent in ML development. (ii) Certain ML pipeline stages, particularly data preprocessing and model training, contain significantly higher SATD density than validation or deployment stages, suggesting that early-phase ML code (e.g., data wrangling, feature engineering, training loops) is especially prone to debt. (iii) SATD is introduced earlier in ML project timelines, reflecting the exploratory nature of ML coding. (iv) Long-lived SATDs in ML code are typically introduced during large, multi-file changes, though these changes often exhibit low code complexity.

3.1.3 Enterprise Software

Zampetti et al. (2021) explored how SATD practices differ between industrial software development and open-source projects, via a mixed-method approach. The

authors first conducted 10 semi-structured interviews with software professionals to gather insights, then surveyed 101 developers (52 from industry, 49 from OSS) about their habits and attitudes in writing SATD comments. They inquired about how often developers mark debt in code, what they include in such comments, and whether there are organizational policies around it. Surprisingly, the core practices of annotating technical debt in code were very similar in industry and open-source contexts.

However, the study uncovered a cultural divergence: in open-source projects, it is common and broadly acceptable to leave `TODO` and `FIXME` notes in the code (indeed many OSS contributors treat it as normal documentation of technical debt). In contrast, in corporate environments, explicit SATD comments may be discouraged by organizational guidelines or team culture. Some companies have policies to routinely remove or address TODOs before releases, and developers reported hesitancy to admit shortcomings in code comments for fear of blame or a perception of negligence. In fact, while open-source code is often littered with self-admitted debt, at some companies such comments are either not allowed or are expected to be transitory. The survey also found that management oversight plays a role: industrial developers sometimes only add SATD comments if there is a tracking mechanism (like an associated JIRA ticket), whereas OSS developers freely use comments as personal notes.

Li et al. (2023b) conducted a comprehensive industrial case study of SATD at a large embedded systems company. It combined repository mining with practitioner interviews to understand SATD from developers' perspectives in a closed-source, safety-critical context. The researchers collaborated with an industry partner and

mined multiple artifact sources in the company's projects, including issue tracker entries, source code comments, and commit messages, to identify any self-admitted debt. They applied an automated SATD detection approach (based on their prior multi-source classifier) to flag SATD items across these sources. Then, they conducted 12 in-depth interviews with software engineers, architects, and project managers at the company to discuss the identified SATD and general attitudes toward technical debt.

The study yielded rich qualitative insights. (i) It characterized the core properties of SATD in industrial projects: for example, many SATD comments were found in test code and workaround modules, and developers tended to use company-specific acronyms rather than generic `TODO` tags. (ii) It captured developers' attitudes: some saw SATD comments as helpful memory joggers, while others viewed them as clutter to be avoided, showing a split in culture even within the firm. (iii) It identified common triggers for introducing SATD (tight deadlines, hardware constraints forcing quick hacks) and triggers for repaying SATD (impending product release, or when debt causes a bug). (iv) It examined relationships between SATD in different sources: e.g., a developer might file a ticket about a refactoring (issue tracker SATD) instead of writing a code comment, indicating some debts were tracked formally rather than in code. (v) It documented current practices for managing SATD: the company had informal guidelines like "fix TODOs before final testing" and some teams held periodic debt review meetings, but there was no systematic tool in place. (vi) It surfaced challenges: interviewees mentioned difficulty in tracing debt (no unified view), fear of admitting certain issues due to compliance processes, and a lack of tool support integrated in their IDE for debt tracking.

3.1.4 Blockchain Infrastructure

Qu et al. (2022a) conducted an empirical study to investigate the types, prevalence, and detection challenges of SATD in blockchain software systems. The study addresses the identification and classification of SATD, focusing on understanding how SATD manifests in blockchain software and how its detection differs from traditional software. The dataset consists of six open-source blockchain framework projects: Bitcoin, Ethereum (go-ethereum), Diem, Solidity, Hyperledger Fabric, and Chia Blockchain, spanning multiple domains (cryptocurrency, smart contracts, distributed ledger). Importantly, the study focuses on SATD in the framework and infrastructure code of these blockchain systems (e.g., node implementations, compilers, distributed ledger platforms), rather than in smart contract code itself. The authors extracted 144,815 comments, manually labeled SATD vs. non-SATD using an expanded taxonomy of seven SATD types: design, defect, requirement, test, documentation, compatibility, and a newly proposed "resource debt", specific to blockchain contexts (e.g., resource allocation and network latency). Identification is fully manual in this study; no machine learning models were used, though the authors recommend future use of pre-trained models due to limitations of task-tag-based methods. The study reports that SATD is prevalent across all projects (SATD proportions range from 1.7% to 28.4% of comments) and that defect debt and design debt dominate. The authors also observe that task tags (e.g., `TODO`, `FIXME`) are unreliable SATD indicators in blockchain frameworks, in contrast to prior findings in Java systems.

Following their previous study, Qu et al. (2022b) investigated the automated detection of SATD in blockchain software projects, addressing both domain-specific

challenges and methodology adaptation. The dataset consists of six popular open-source blockchain projects from their prior study. Then, they used a fine-tuned BERT model with cost-sensitive training to handle class imbalance, and compared its performance against the MAT baseline (Guo et al. 2019). Features are purely semantic, learned by the BERT model; no manual patterns or traditional NLP features are used. Evaluation uses cross-project experiments, reporting precision, recall, and F1. The BERT-based approach achieved a 102% F1-score improvement over MAT, with statistically significant results.

Pinna et al. (2023) conducted an empirical study examining the presence of SATD in open-source blockchain projects and comparing its prevalence with that in traditional open-source software. The study explores both SATD identification and its potential impact on the maintainability and trustworthiness of blockchain systems, where unmanaged SATD may pose particular risks. The dataset consists of ten popular open-source blockchain framework projects (Bitcoin, Ethereum, USD Coin, Binance, Terra, XRP, Polkadot, Cardano, Avalanche, and Solana), selected based on market capitalization and popularity. A blockchain framework, as defined in this study, refers to foundational software systems (like Ethereum, Bitcoin, Diem, etc.) that implement the core blockchain infrastructure, including networking, consensus, smart contract execution, and associated APIs. Thus, analysis focuses on framework and infrastructure code, not smart contract code. The authors extracted and preprocessed 191,533 comments and applied an NLP-based SATD detection pipeline using the Stanford maximum entropy classifier, trained on Maldonado et al. (2017a)'s dataset, targeting two SATD types: design debt and requirement debt.

Results indicate that SATD prevalence in blockchain projects (mean 8.35% of comments) is notably higher than in a reference set of Java open-source projects (mean 5.52%), with design debt occurring approximately four times more frequently than requirement debt. Additionally, blockchain projects exhibited a more even distribution of SATD across files, though high concentrations in certain projects (e.g., Bitcoin, XRP, Polkadot) may jeopardize long-term maintainability and trust.

3.1.5 Our Choice of SATD Detector Tool

Based on a literature review, we evaluated SATD detection tools available as of 2020, marking the start of our first study (Chapter 4). Table 3.1 provides an overview of the candidate SATD detector tools we considered. To select an appropriate SATD detector, we studied and compared multiple criteria across several prior studies proposing SATD detection approaches.

First, the availability of a replication package (i.e., either the source code of the proposed approach or a list of SATD indicator patterns) was a primary selection criterion. This is because re-implementing existing approaches can be complicated, error-prone, and time-consuming. Out of thirteen studies, seven released their replication packages (Potdar and Shihab 2014; de Freitas Farias et al. 2015; Bavota and Russo 2016; Huang et al. 2017; Liu et al. 2018; Freitas Farias et al. 2020; Yu et al. 2020). We excluded the remaining six studies (S. Maldonado and Shihab 2015; Maldonado et al. 2017a; Flisar and Podgorelec 2019; Ren et al. 2019; Wattanakriengkrai et al. 2019; Maipradit et al. 2020) due to the lack of available replication material. Next, we considered the purpose of the tools. Since our focus is on detecting SATD as a binary classification task, we excluded tools designed to

identify specific SATD types. This led us to exclude three additional methods (de Freitas Farias et al. 2015; Bavota and Russo 2016; Freitas Farias et al. 2020), leaving four methods (Potdar and Shihab 2014; Huang et al. 2017; Liu et al. 2018; Yu et al. 2020) that perform SATD versus non-SATD classification. Our next criterion focused on the detection approach. These four tools can be grouped into three categories: (i) NLP-based approaches, (ii) machine learning (ML)-based approaches, and (iii) hybrid approaches combining NLP and ML. Below, we discuss each category and provide our rationale for selecting or excluding each approach.

Potdar and Shihab (2014) developed a curated list of 62 SATD indicator patterns. While such patterns can be applied for SATD detection, fully automated detection based solely on these patterns tends to result in a high false positive rate. For example, Bavota and Russo (2016) used a similar list of patterns and reported that over 25% of their manually investigated SATD comments were false positives. Considering this finding, and the highly descriptive nature of smart contract comments, we opted not to employ this technique. Other researchers have also highlighted the high false positive rate of pattern-based approaches; for instance, Freitas Farias et al. (2020) reported that approximately 40% of SATD items detected using de Freitas Farias et al. (2015)'s indicator patterns were false positives.

Huang et al. (2017) proposed a voting mechanism based on training multiple ML sub-classifiers. Liu et al. (2018) later implemented this approach as a plugin integrated into the Eclipse development environment. However, we chose not to adopt this tool, as Yu et al. (2020) noted that several comment instances were clearly misclassified in Liu et al. (2018)'s study, which could negatively impact the resulting classification model.

Yu et al. (2020) introduced a hybrid two-step framework called Jitterbug. In the first phase, they applied a novel pattern recognition algorithm to automatically detect ”easy-to-find” SATD comments. In the second phase, they implemented a semi-automated continuous learning technique to identify “hard-to-find” SATD comments without indicator patterns. We decided to use only the “easy-to-find” detection component for three reasons. First, our work prioritizes precision over recall, and this component achieved near-100% precision in detecting SATD comments across ten open-source projects, outperforming prior baselines such as CNN (Ren et al. 2019) and MAT (Guo et al. 2021). We further evaluate the performance of this component in the smart contracts context in Appendix A.1. Second, the precision of the “hard-to-find” SATD detector (the second step of the framework) has not been reported, and evaluating this component is outside the scope of our study. Our goal is to reuse a reliable SATD detector rather than develop a new one. Finally, due to the semi-automated nature of the “hard-to-find” component, it was not feasible to apply it to our dataset, which contains over 17 million comments.

Table 3.1: Overview of the candidate SATD detector methods (2014–2020).

Study	Year	Venue	Approach	Automation Level	Debt Type	Replication Package Available.?
Potdar and Shihab (2014)	2014	ICSME	NLP	Automatic	SATD, Non-SATD	✓
de Freitas Farias et al. (2015)	2015	ICSME	NLP	Automatic	Architecture, Build, Code, Defect, Design, Documentation, Infrastructure, People, Process, Requirement, Service, Test	✓
S. Maldonado and Shihab (2015)	2015	MSR	NLP	Manual	Design, Defect, Doc, Requirement, Test	
Bavota and Russo (2016)	2016	ICSE	NLP	Automatic	Code, Defect, Requirement, Design, Test, Documentation	✓
Maldonado et al. (2017a)	2017	TSE	ML	Automatic	Design, Requirement, Non-SATD	
Huang et al. (2017)	2018	EMSE	ML	Automatic	SATD, Non-SATD	✓
Liu et al. (2018)	2018	ICSE	ML	Automatic	SATD, Non-SATD	✓
Flisar and Podgorelec (2019)	2019	JSEP	ML	Automatic	SATD, Non-SATD	
Wattanakriengkrai et al. (2019)	2019	APSEC	ML	Automatic	Design, Requirement	
Ren et al. (2019)	2019	TOSEM	DL	Automatic	SATD, Non-SATD	
Freitas Farias et al. (2020)	2020	EMSE	NLP	Semi-automatic	Architecture, Build, Code, Defect, Design, Documentation, People, Requirement, Test	✓
Maipradit et al. (2020)	2020	SANER	ML	Automatic	On-hold SATD	
Yu et al. (2020)	**2020**	**TSE**	**ML**	**Semi-automatic**	**SATD, Non-SATD**	✓

Summary

Research on Self-Admitted Technical Debt (SATD) has evolved through distinct methodological phases, from early pattern-based NLP and machine learning to advanced deep learning and transformer models. While these approaches have achieved high performance on benchmark datasets, they exhibit significant limitations in generalizability and practical application. Pattern-based and simple ML methods struggle with recall and project-specific vocabularies, while even state-of-the-art models are predominantly trained and validated on a limited set of open-source Java projects. This raises concerns about their effectiveness in specialized domains like blockchain, where comment conventions and debt types may differ. Furthermore, the heavy reliance on supervised learning creates a dependency on large, manually labeled datasets, which are scarce for many domains. Critically, many advanced models have been shown to effectively relearn simple patterns captured by basic heuristics like matching annotation tags (e.g., `TODO`, `FIXME`, `HACK`, etc.), calling into question the reported magnitude of improvement over simpler baselines.

These gaps are particularly acute for analyzing smart contracts, a domain with distinct development practices and a lack of large, labeled SATD datasets. Therefore, for our large-scale study of SATD in Solidity code (Chapter 4), we select and employ the Jitterbug framework's high-precision pattern-based component. This choice prioritizes accuracy and practicality over recall, providing a reliable and automatable method for identifying a foundational corpus of "easy-to-find" SATD without the need for domain-specific model training or extensive manual validation.

3.2 Smart Contract Maintenance

Bosu et al. (2018) surveyed 156 Blockchain software (BCS) developers from 145 GitHub projects. They found fundamental differences between BCS and traditional SE practices, particularly the emphasis on security, immutability, decentralization, and the challenges posed by an immature ecosystem and contract maintenance difficulties. These factors necessitate specialized tools, such as enhanced testing

frameworks, secure development environments, custom IDEs, smart contract debuggers, formal verification techniques, and static analysis capabilities.

Chen et al. (2021c) conducted the first in-depth empirical study focusing specifically on the maintenance-related concerns of post-deployed Ethereum smart contracts. Unlike traditional software, smart contracts on Ethereum are immutable once deployed, posing novel challenges to software maintenance. The study frames its analysis using the ISO/IEC 14764 classification of maintenance into four categories, corrective, adaptive, perfective, and preventive, and explores the unique difficulties smart contract developers face in each. More specifically, corrective maintenance is hindered by the lack of mature debugging tools and weak community support (Zou et al. 2021; Norvill et al. 2017; Bosu et al. 2018), despite the existence of academic tools that are often impractical in real-world scenarios. Adaptive maintenance is challenged by unreliable callee contracts, which can crash or `selfdestruct`, leading to cascading failures (Fröwis and Böhme 2017). Perfective maintenance faces Solidity and EVM scalability limitations, complicating feature expansion and gas optimizations (Zou et al. 2021; Bez et al. 2019). Preventive maintenance is constrained by the ecosystem's immaturity, lack of standardization, and insufficient reference code, making advanced techniques like code smell removal (Arcelli Fontana et al. 2016) and technical debt detection (Yan et al. 2019) difficult.

In addition to mapping the problem space, the study categorizes current maintenance strategies into four main approaches. First, offline verification techniques, such as static analysis, fuzz testing, and formal verification, are widely used to identify bugs and vulnerabilities before deployment. Second, online verification

methods provide runtime protection by detecting malicious inputs or enforcing security policies during contract execution. Third, emergency destruction mechanisms leverage the `selfdestruct` opcode to remove vulnerable contracts and transfer remaining funds, though this undermines immutability and user trust. Fourth, contract evolution through `delegatecall` enables developers to redirect execution to new logic contracts, preserving the contract's address and storage layout. This proxy-based approach, while powerful, introduces complexities related to security, transparency, and standardization.

Despite the availability of these techniques, the study underscores that each method carries significant trade-offs. Offline tools are often too academic or difficult to integrate into real workflows; online methods require deep EVM integration; `selfdestruct` risks trust and asset loss; and upgradeable proxy contracts demand careful design to avoid security pitfalls. These limitations reveal a pressing need for more usable, standardized, and trustworthy maintenance solutions, particularly as smart contract ecosystems continue to scale and diversify.

Both Bosu et al. (2018) and Chen et al. (2021c) underscored the prevalence of TD in smart contracts. Bosu et al.'s survey revealed that blockchain developers not only frequently encounter TD, but also spend significantly more effort addressing it compared to developers of traditional software systems (Bosu et al. 2018). Similarly, Chen et al. (2021c) emphasized the lack of specialized SE practices and publicly available datasets as major challenges for studying TD in this domain. Yet, there remains gaps in demystifying TD characteristics, as well as in the availability of empirical data and actionable insights to support end-users and developers in detecting, reasoning about, avoiding, or addressing TD in smart contracts.

Expanding on Chen et al. (2021c)'s work, Rosa et al. (2025) conducted an empirical study to understand the motivations and practices behind smart contract maintenance. Despite the immutability of deployed smart contracts, maintenance activities remain prevalent due to the need for bug fixes, feature upgrades, and code quality improvements. Analyzing 590 commits across 14 popular open-source Solidity repositories, the authors present two taxonomies: one categorizing the reasons for changes and another identifying the patterns of modifications. The study finds that perfective maintenance (e.g., refactoring and code readability improvements) accounts for the largest share (39%), followed by corrective maintenance (32%), adaptive changes (22%), and preventive efforts (e.g., adding tests) at 7%. Surprisingly, refactoring is the most common reason for changes, even more frequent than bug fixing, highlighting developers' focus on code quality and maintainability. Additionally, aesthetic and structural changes (such as adding comments or restructuring functions) are as prevalent as logic-altering modifications. The authors argue that current tools for smart contract development, especially for testing, refactoring, and automated bug detection, are either underused or insufficient. Their findings underscore the need for better maintenance-support tools tailored to smart contract environments.

Antonino et al. (2022) presented a systematic survey of formal specification and verification approaches for smart contracts. Given the high-stakes, immutable nature of smart contracts, they argue that rigorous formal methods are essential for ensuring correctness and mitigating post-deployment vulnerabilities. The authors analyzes over 40 academic studies and tools, classifying them by their underlying specification methods (e.g., temporal logic, finite automata) and verification

techniques (e.g., model checking, symbolic execution, and theorem proving). The authors find that most verification efforts prioritize safety properties, such as preventing reentrancy or arithmetic overflows vulnerabilities, while liveness and behavioral correctness are comparatively underexplored. They also identify a critical gap between theoretical formalism and practical usability: many tools demand deep formal methods expertise and do not integrate well with existing smart contract development workflows. To bridge this gap, the study emphasizes the importance of domain-specific languages (DSLs) and intermediate representations (e.g., control flow graphs) that facilitate model construction and improve automation.

Importantly, the authors highlight the proxy pattern, a common mechanism for enabling smart contract evolution, as a major challenge for formal verification. The pattern introduces execution indirection via `delegatecall`, causing the proxy contract to dynamically execute the logic of another contract. This indirection complicates verification because the control flow and state updates are split across contracts. As a result, verifying the proxy in isolation is insufficient; one must reason about the composition of both proxy and logic contracts. However, most current tools lack the ability to perform cross-contract analysis or support dynamic dispatch. The authors stress that while the proxy pattern enhances flexibility, it introduces significant verification obstacles, particularly for tools that assume self-contained, statically linked contract logic.

Wöhrer and Zdun (2018) Wöhrer and Zdun explores common design patterns used in Ethereum smart contracts, with the goal of improving their correctness, security, and maintainability. Using a Multivocal Literature Review and qualitative

methods, the authors identified 18 design patterns, grouped into five categories: Action and Control, Authorization, Lifecycle, Maintenance, and Security.

While the authors briefly covers all pattern categories, it places particular importance on Maintenance patterns, which address the challenge of smart contract immutability. Data Segregation separates storage from logic, allowing new logic to be applied to persistent data. Satellite pattern outsources functionalities to external contracts, which can be independently updated. Contract Relay (i.e., proxy pattern) provide mechanisms for version control by acting as a proxy that forwards calls to the current version. However, the authors also highlight certain risks associated with Maintenance patterns. Proxy-based approaches like Contract Relay require strict consistency in data storage layouts, or else they risk corrupting state and introducing unintended behavior. Moreover, delegating logic to external contracts increases complexity and potential security dependencies. There are also trust concerns, as users must rely on contract owners not to abuse contraction evolution mechanisms. Despite these risks, Maintenance patterns are seen as essential tools for managing long-term smart contract functionality in evolving ecosystems.

M. and Kaganurmath (2022) provided review of existing contract evolution mechanisms for Ethereum smart contracts, highlighting the inherent tension between blockchain immutability and the evolving requirements of decentralized applications. The authors categorize upgrade patterns into two main groups: data segregation and proxy-based approaches. Data segregation patterns, including basic segregation, satellite contracts, and register-based models, decouple logic from

storage to enable partial upgrades by redeploying logic components while preserving contract state. However, these patterns do not support the evolution of the data contract itself and lack built-in protections against known vulnerabilities.

In contrast, proxy-based patterns, such as inherited storage, eternal storage, and unstructured storage proxies, enable full contract evolution through the use of `delegatecall`, allowing new logic implementations to operate on the same storage. While more powerful, these approaches introduce complexity related to storage alignment, security, and scalability. Notably, the authors observe that none of the existing patterns effectively address security risks such as reentrancy or DAO-style attacks. To address these limitations, the authors introduce the Comprehensive-Data-Proxy Pattern, a hybrid design that combines data segregation with proxy-based delegation and incorporates an additional authentication and verification layer. This architecture enables full upgrades of both logic and storage components while enhancing resilience against common attacks. Experimental evaluations demonstrate that the proposed approach achieves comparable performance to existing methods, with improved security guarantees.

3.3 Proxy Contract Detection

Feist et al. (2019) present Slither, a static analysis framework for Solidity smart contracts, designed to detect vulnerabilities, identify optimization opportunities, and improve code understanding. Slither converts Solidity code into an intermediate representation called SlithIR, which uses Static Single Assignment (SSA) form to enable advanced static analyses like dataflow and taint tracking. It includes over 20

built-in detectors for common vulnerabilities such as reentrancy, uninitialized variables, and access control flaws. In addition, Slither provides specialized checks for upgradeable contracts using the `delegatecall` proxy pattern. These checks detect critical issues such as storage layout mismatches between proxy and logic contracts, function selector collisions, improper or missing initializer calls, and variable order inconsistencies between contract versions. This functionality helps developers avoid subtle bugs that could lead to storage corruption or unauthorized logic replacement during upgrades.

Salehi et al. (2022) examine the widespread use of contract evolution mechanisms in Ethereum, challenging the assumption that smart contracts are immutable once deployed. While immutability enhances trust and transparency, the authors show that developers frequently implement upgrade paths to patch vulnerabilities or add features, often at the cost of decentralization and security. They classify evolution mechanisms into retail (e.g., parameter changes) and wholesale patterns (e.g., contract migration, `CREATE2`-based metamorphosis, and `call`-based data separation and `delegatecall`-based proxy contracts). `delegatecall`-based proxy contracts emerge as the most popular due to their efficiency and minimal code changes. Analyzing over 1.4 million proxy contracts deployed between September 2020 and July 2021, including three variants (i.e., regular, UUPS, and beacon). Governance analysis reveals that 48% are controlled by a single externally owned account (EOA), and only a minority use multi-signature or decentralized governance. The study concludes that Ethereum's immutability is often bypassed, raising critical concerns about the actual decentralization and security of many so-called "decentralized" applications.

Qasse et al. (2023) an exploratory study on the smart contracts contract evolution on the Ethereum blockchain, aiming to examine how frequently upgrade patterns are used, the reasons behind contract upgrades, and their effects on contract activity. Recognizing that smart contracts are immutable by design, the authors focus on community-driven mechanisms, such as proxy patterns and data separation, that enable code upgrades while preserving state. By analyzing data from Ethereum ETL and Etherscan, they identify 27,000 proxy contracts, of which 51% are upgradeable, yet only 32% of a sample set were actually upgraded. Using tools like Git diff and SmartBugs, they classify upgrade motivations into bug fixes, feature additions, and gas optimizations, and evaluate their impact through transaction data.

Bodell III et al. (2023) present a large-scale empirical study on proxy-based upgradeable smart contracts (USCs), focusing on their prevalence, implementation patterns, and security implications. Recognizing that USCs offer flexibility to update immutable blockchain contracts but also introduce significant security risks, the authors develop USCHUNT, a static analysis framework to detect and analyze USCs using both syntactic and semantic features. Analyzing over 861,000 contracts across eight major blockchains, they identify 8,815 USCs and uncover 11 unique USC design patterns. Their findings highlight that USCs collectively hold over $3 billion in digital assets, but also suffer from 2,546 real-world security and safety issues, including storage layout clashes, function selector collisions, insufficient upgrade compatibility checks, and risky upgrade policies. The study contributes a detailed taxonomy of USC behaviors, exposes widespread vulnerabilities, and emphasizes the urgent need for secure upgrade practices in decentralized applications.

Meisami and III (2023) present a comprehensive survey of upgradeable smart contract patterns, with a particular focus on proxy-based designs used on the Ethereum blockchain. Acknowledging the immutability of smart contracts and the critical need for secure upgrades due to evolving threats and software bugs, the authors catalog various proxy patterns such as Transparent Proxy, UUPS, and Diamond Proxy. They analyze over 64,000 verified contracts using the Slither static analysis tool, refining its proxy detection algorithms to reduce false positives by 70%. The authors highlight key challenges like storage layout clashes, function selector collisions, and Solidity version incompatibilities, proposing algorithmic improvements to more accurately identify upgradeable proxy contracts.

3.3.1 Comparative Analysis of Proxy Contracts Detection Methods

Among the prior works discussed in Section 3.3, four studies (Feist et al. 2019; Salehi et al. 2022; Bodell III et al. 2023; Meisami and III 2023) have proposed techniques for detecting proxy contracts. In the following, we compare these studies in detail based on the characteristics of their proxy detection methods. Table 3.2 shows a summery of our comparison.

Table 3.2: Comparison of studies on proxy detection. Blank cells indicate missing or unspecified information in the corresponding studies.

Study	**Proxy detection characteristics**									
	Detection approach		Proxy signatures		Analysis level		Proxy function		Usage status	
	Dynamic	Static	Property #1	Property #2	Source Code	Bytecode	Fallback Function	Any Function	Active	Inactive
Our Method (Chapter 5)	*		*	*				*	*	
Bodell III et al. (2023)		*	*		*		*		*	*
Qasse et al. (2023)		*	*		*			*	*	*
Salehi et al. (2022)	*		*	*			*		*	
Feist et al. (2019)		*	*		*		*		*	*

1. Detection Approach. This aspect concerns how a study identifies proxy contracts, either through run-time dynamic analysis or static analysis of the contract's source code or bytecode.

Salehi et al. (2022) adopt a dynamic approach by replaying past transactions. This means that the authors re-executed past transaction, using the same inputs (e.g., `calldata`, sender, gas, value), to observe or analyze the behavior of a smart contract as it occurred on-chain. A key limitation of dynamic analysis, particularly when based on replaying transactions, is its reliance on the availability and completeness of historical transaction data. If a contract is new, infrequently used, or lacks representative execution history, replaying transactions may not expose its proxy behavior, leading to incomplete detection. More importantly, replaying transactions is computationally expensive and error-prone, as it requires reconstructing

historical blockchain state and simulating full EVM execution for each transaction. This overhead limits scalability in large-scale studies.

In contrast, Bodell III et al. (2023), Qasse et al. (2023), and Feist et al. (2019) rely on static analysis, inspecting the source code without executing the contract. Static analysis offers broader coverage in terms of contract types, as it can be applied regardless of transaction activity. However, its effectiveness hinges on the availability of source code, which is often unavailable, obfuscated, or incomplete. Prior research has consistently shown that a significant portion of Ethereum smart contracts are not verified on-chain (Section 2.3), making source code–based analysis impractical for many contracts. In contrast, dynamic detection methods, which do not rely on source code, are better suited for analyzing all contracts, including those lacking verified sources.

2. Proxy Signatures. This criterion defines the conditions under which a contract qualifies as a proxy contract. Specifically, a contract is considered a proxy if it satisfies the two inherent properties described in Section 2.15.4: (1) delegating at least one function call via `delegatecall`, and (2) preserving the interface of the delegated function.

Among prior works, only Salehi et al. (2022) evaluate both properties, resulting in a more accurate detection approach. In contrast, Bodell III et al. (2023) consider only Property #1 by checking for the presence of a `delegatecall` instruction within the control flow graph of the `fallback` function. While implementing `delegatecall` within the `fallback` function is typically a sign of proxy contracts, it can result in false positive if Property #2 left unchecked. In Listing 3.1, we provide an illustrative real world example where a contract uses `delegatecall` within its

fallback function but fails to preserve the interface of the logic contract. As such, it behaves as an adapter contract rather than a true proxy. Specifically, an adapter modifies the interface of the object it represents to ensure compatibility with the client. This is evident from the developers' self-admission in lines 2 to 5.

Qasse et al. (2023) use a regular expression to search for the keyword `delegatecall`, and Feist et al. (2019) similarly rely on its textual presence. While these keyword-based approaches are lightweight and scalable, they are prone to false positives. This limitation stems from the fact that not every `delegatecall` operation is indicative of a proxy contract. For example, keywords like `delegatecall` can appear in comments or unused or dead code, especially since smart contracts tend to have higher comment density than typical software. Another common non-proxy use case of `delegatecall` occurs in contracts that internally route calls to other contracts to achieve modularity or extensibility. Although such contracts leverage `delegatecall` to pass execution to target logic, they do not fulfill the proxy contract criteria, as they typically perform additional pre-processing and do not maintain a strict one-to-one interface mapping.

```
[...]
/**
 * @dev This code allows us to redirect pre-signed calls with different
 * function selectors to our own.
**/
function () public
{
   bytes memory calldata = msg.data;
   bytes4 new_selector = standardSigs[msg.sig]; // target function selector is changed.
   require(new_selector != 0);
   assembly {
     mstore(add(0x20, calldata), new_selector)
   }
   require(address(this).delegatecall(calldata));
   [...]
   }
}
[...]
```

Listing 3.1: The SiuCoin's fallback function that implements an adapter contract.

3. Analysis Level. This criterion reflects whether the detection approach operates at the source code or bytecode level, and applies specifically to static analysis methods. Source code refers to the human-readable form of a contract, while bytecode is the compiled, machine-readable format executed by the Ethereum Virtual Machine (EVM). Among the reviewed studies, Bodell III et al. (2023), Qasse et al. (2023), and Feist et al. (2019) perform their analyses at the source code level.

As discussed in Section 5.1, the majority of Ethereum smart contracts are not verified on Etherscan. Consequently, approaches that rely on source code are limited in applicability when source code is unavailable. Even when source code is accessible, it may include noise such as unreachable code or unused contracts, which can negatively impact the accuracy of detection tools (Section 6.8.1).

Notably, none of the existing studies reviewed perform proxy detection directly at the bytecode level. Bytecode-level detection circumvents the need for verified source code and could potentially enable broader coverage. However, it introduces

additional complexity, particularly the need to decompile and analyze low-level EVM instructions.

4. Proxy Function. This criterion identifies where within a contract the proxy functionality is implemented. While proxy logic is typically placed in the `fallback` function, it can technically reside in any function, as long as the function containing the `delegatecall` has a signature that matches the one receiving the delegated calls (i.e., Property #2).

Salehi et al. (2022), Bodell III et al. (2023), and Feist et al. (2019) restrict their analysis to fallback functions, which are traditionally used for implementing proxy behavior. In contrast, Qasse et al. (2023) extend their analysis to all functions within a contract by searching `delegatecall` keyword, allowing them to detect non-traditional proxy contracts that implement delegation logic outside the fallback function. This broader scope increases the chances of identifying less conventional proxy patterns.

5. Usage Status. This criterion distinguishes the types of proxy contracts a method can detect, specifically, whether it identifies active or inactive proxy contracts. An active proxy contract is one that has invoked its proxy functionality at least once after it is deployed. In contrast, an inactive proxy has never used its delegation logic following deployment. Salehi et al. (2022) are limited to detecting active proxy contracts, as they rely on transaction replaying, which restricts their coverage to contracts that have exercised their proxy functionality on-chain. In comparison, Bodell III et al. (2023), Qasse et al. (2023), and Feist et al. (2019) can detect both active and inactive proxy contracts, since their static analysis methods examine contract code rather than requiring evidence of execution.

Summary

Existing proxy detection methods exhibit notable shortcomings across multiple dimensions. Dynamic methods, such as transaction replaying, are computationally expensive, error-prone, and limited to detecting active proxies with observable transaction histories. Static approaches often rely on verified source code, which is unavailable for most smart contracts, and are susceptible to unreachable or misleading code. Furthermore, several studies only partially enforce proxy-defining properties, particularly neglecting Property #2, which can lead to false positives, such as adapter contracts. Additionally, most approaches restrict detection to the `fallback` function, overlooking delegation logic implemented in other functions. Importantly, none of the reviewed studies directly utilize raw bytecode or existing public transaction records for detection.

To address these gaps, our proposed approach (Section 5.4) analyzes publicly available transactional data without requiring source code, full bytecode parsing, or transaction replaying, and enforces both proxy-defining properties, offering a practical and scalable alternative for detecting active proxy contracts.

3.3.2 Comparative Analysis of UPC and FPC Detection Methods

We also examine the four most recent studies in the field (Salehi et al. 2022; Bodell III et al. 2023; Qasse et al. 2023; Feist et al. 2019) that propose methods for distinguishing Upgradeable Proxy Contracts (UPCs) from Forwarder Proxy Contracts (FPCs) (Section 2.15.5). Our analysis is structured around three key criteria, as outlined below. In the following, we present a detailed comparison based on each of these criteria. Table 3.3 shows a summery of our comparison.

Table 3.3: Comparison of studies on proxy type classification.

Study	Detection Approach		Analysis Level	UPC Pattern							
	Dynamic	Static	Bytecode / Source	Beacon	Registry	ERC-1822 Universal	ERC-2535 Diamonds	Transparent	ERC-1967	Inherited Storage	Eternal Storage
Our Method (Chapter 6)	*	*	Bytecode	*	*	*	*	*	*	*	*
Bodell III et al. (2023)	*	*	Source	*	*	*	*	*	*	*	*
Qasse et al. (2023)	*	*	Source			*	*	*			
Salehi et al. (2022)	*	*	Bytecode	*		*		*			
Feist et al. (2019)		*	Source								

1. Detection approach. The first criterion refers to the methodology utilized for identifying upgradeable proxy contracts. This can be achieved either through static, dynamic, or a hybrid method. The dynamic approach involves analyzing smart contract transactions, while the static method involves running an analyzer on the smart contract source or bytecode. In the given studies, Salehi et al. (2022), Bodell III et al. (2023), and Qasse et al. (2023) use both dynamic and static methods albeit to different extent. On the other hand, Feist et al. (2019) rely solely on static methods.

More specifically, Salehi et al. (2022) used a dynamic method to extract the external dependencies, then applied a static analyzer to further validate if the given contract implements an upgrade function. Bodell III et al. (2023) undertook the task of running a static analyzer on the proxy's source code, applying advanced techniques such as context-sensitive, flow-insensitive, and cross-contract data-flow analysis. Similarly, this study developed an "explorer querying" component to retrieve a contract's dependencies, embed related contracts into the same compilation

unit, before running their cross-contract analysis. However, this approach is limited in scenarios where the proxy has multiple implementation contracts, one of which handles the upgrade mechanism, necessitating more advanced cross-contract analysis to account for multiple implementations (Bodell III et al. 2023).

Qasse et al. (2023) employ an innovative approach that leverages the structural definitions of Upgradeable Proxy Contract (UPC) patterns to distinguish between UPCs and FPCs. Their method identifies certain pattern-specific features (e.g., keywords) in the contract's source code (e.g., the presence of the `Proxiable` or `IDiamond` interfaces) to determine its classification. If a detected proxy does not conform to the expected features of these patterns, it is classified as adhering to the Transparent UPC Pattern. For example, in the context of the ERC-1822 Universal UPC Pattern, the authors extracted the proxy's `delegatecall` transactions and dynamically obtained the corresponding implementation contract address (i.e., callee address field). They then verified whether the implementation contract adhered to the `Proxiable` interface. while relying on the unique structural features of upgradeability patterns is indeed a promising strategy, it does not necessarily ensure that the contract actually implements an upgrade function. In contrast, methods proposed by Salehi et al. (2022) and Bodell III et al. (2023), only classify a proxy as upgradeable if they explicitly identify at least one public or external function (e.g., an upgrade function) capable of replacing the current implementation contract with a new one, potentially resulting in higher classification precision.

Feist et al. (2019) introduced Slither, a static analysis tool designed to identify security vulnerabilities and coding issues in smart contracts. As noted by Bodell III et al. (2023), Slither includes native support for detecting upgradeable proxy

contracts. However, its capabilities have certain limitations. Slither's approach primarily relies on searching for keywords such as "upgradeable" and "proxy," as well as identifying the presence of `delegatecall` within fallback functions. Additionally, this method lacks the ability to analyze cross-contract interactions, which significantly restricts its effectiveness in detecting complex UPC patterns.

In addition, none of the existing methods are capable of identifying UPCs when the upgrade functions are implemented in transitive dependencies (Section 6.2.4.2.2 and Section 6.2.4.2.3). For example, in the ERC-1822 Universal UPC Pattern, if the proxy's implementation contract is itself a proxy, the upgrade logic may reside within the implementation contract of that implementation contract. In such cases, the upgrade path becomes multi-layered, and the actual upgrade function is located deeper in the dependency chain, making it difficult for current methods to detect the presence of upgradeability mechanisms accurately.

2. Analysis level. The second criterion pertains to static detection methods and distinguishes between analyses performed on a smart contract's source code versus its bytecode. In this comparison, Bodell III et al. (2023), Qasse et al. (2023), and Feist et al. (2019) operate at the source code level, whereas Salehi et al. (2022) conduct their analysis on the bytecode. Similar to proxy detection techniques, methods relying on source code analysis are constrained to verified contracts and may be error-prone, as the source code can include unreachable or dead proxy code (Section 6.8.1).

3. UPC pattern. The third criterion evaluates the specific UPC patterns that each study is capable of detecting, including the eight patterns introduced in Section 2.15.6. A study is marked as capable of detecting a particular pattern if there

is any indication, explicit or implicit, that the detection of that pattern is supported. Most studies cover a wide array of these patterns, with Bodell III et al. (2023) and UPC Sentinel supporting all eight. In contrast, Salehi et al. (2022) and Qasse et al. (2023) omit several important patterns, and Feist et al. (2019) appears to detect only the Transparent UPC pattern, lacking support for the others.

Summary

Existing proxy type classification methods exhibit notable limitations in both breadth and depth. Source code–based approaches are restricted to verified contracts and are vulnerable to unreachable or misleading code, while bytecode-level methods often fail to capture the full range of UPC patterns. Some studies enforce only partial classification criteria, frequently neglecting the presence of concrete upgrade functions or relying solely on structural keyword-based indicators, which may not reflect actual upgradeability behavior. Additionally, no existing method adequately handles transitive dependencies, where upgrade logic is implemented across multiple layers of proxy contracts. Furthermore, none of the existing approaches orchestrate static and dynamic techniques in a manner that ensures comprehensive coverage, high precision, and robustness across both verified and unverified contracts.

Critically, proxy type classification is a dependent task that assumes proxies have been correctly detected. Any inaccuracy in the underlying proxy detection step cascades into misclassifications of proxy type (e.g., mislabeling a UPC as an FPC or vice versa). As such, weak proxy detection logic can significantly compromise the reliability of subsequent classification. These limitations collectively highlight the need for more accurate and integrated solutions that address both detection and classification in a unified and scalable manner.

To address these gaps, our proposed approach (Section 6.2) meticulously combines static and multiple forms of dynamic analysis in a layered fashion, operating directly at the bytecode level. It leverages both the inherent structural and behavioral features of known UPC patterns, enabling accurate and comprehensive detection of proxy contracts, classification of proxy types, and identification of specific UPC patterns, all within a unified framework.

CHAPTER 4

Self-Admitted Technical Debt in Ethereum Smart Contracts: A Large-Scale Exploratory Study

4.1 Introduction

Blockchain technology enables digital transactions to be executed in a completely decentralized and secure manner. Due to these characteristics, blockchain is seen as a transformative force capable of reshaping how businesses are conceived and operated. According to Gartner, blockchain could generate up to US$3.1 trillion in new business value by 2030 (Furlonger and Uzureau 2019).

A key player in this landscape is the Ethereum platform, which introduced a significant innovation upon its launch in 2015: support for smart contracts. Smart contracts are computer programs that form the foundation of blockchain-powered

applications, commonly referred to as decentralized applications (DApps) (Oliva et al. 2020). Blockchain platforms that support such applications are often termed programmable blockchains. Ethereum remains the most prominent programmable blockchain, hosting 73.6% of all DApps listed on the State of the DApps website as of April 8, 2022, including 41 out of the top-50 ranked DApps. Popular DApp categories include NFTs (digital collectibles) and DeFi (decentralized finance), underscoring Ethereum's growing influence across business markets.

However, Ethereum's rapid adoption has also triggered a surge in expectations across industry and academia, expectations that frequently exceed the technological maturity of the ecosystem. For example, the prominence of NFTs on Gartner's 2021 Hype Cycle (Gartner 2021), where they reached the Peak of Inflated Expectations, exemplifies the broader hype and the fear of missing out that compels organizations toward hasty adoption (Labazova et al. 2019; Oliva et al. 2020; Lacity and Hoek 2021). This rush can encourage premature development and deployment practices, especially under time pressure.

We conjecture that this environment contributes to the accumulation of technical shortcuts, where developers intentionally opt for quick fixes or suboptimal implementations instead of investing in robust, maintainable solutions. The immaturity of Solidity, Ethereum's dominant smart contract language, exacerbates the issue. Compared to traditional languages like C, Java, or Python, Solidity's ecosystem (e.g., debuggers, static analyzers, and IDEs) is considerably less mature (Chen et al. 2021c), and the scarcity of expert developers further limits access to high-quality training and shared knowledge.

To evaluate this conjecture, we investigate whether smart contracts reflect signs of technical debt (TD), a metaphor introduced by Cunningham (1992) to describe the additional effort incurred by deferring a better solution in favor of short-term gains. Specifically, in line with recent studies in software engineering (Section 3.1), we also focus on a well-established subset of TD known as SATD. SATD refers to instances where developers explicitly acknowledge technical shortcomings, deferred work, or incomplete implementations in the form of source code comments (Potdar and Shihab 2014). These comments represent deliberate trade-offs, signals that the debt is known and consciously accepted during development.

Although prior research has primarily focused on unintentional issues like bugs and vulnerabilities in smart contracts, SATD offers a complementary and equally important perspective (Bosu et al. 2018; Chen et al. 2021c). It captures developer intent and reveals the conscious and intentional compromises made due to constraints such as time pressure, immature tooling, or evolving requirements. Unlike hidden defects that require sophisticated detection, SATD is often explicitly recorded through source code comments, allowing for direct examination of known limitations and rationales behind them.

Studying SATD is essential not only to assess the technical health and maintainability of smart contracts, but also to gain insight into developer reasoning. This is particularly important in the blockchain domain, where deployed contracts are immutable and, in many cases, cannot be upgraded without specialized evolution mechanisms (Section 2.14). As a result, SATD may persist indefinitely, especially when the deferred issue is not urgent enough to justify the complexity or cost of

redeployment, ultimately evolving into a long-term barrier that hampers maintainability, causes miscommunication, or undermines contract functionality.

To investigate our hypothesis, we collect a raw dataset with over 5.7 million smart contracts, extract real-world contracts from it, and study the prevalence and types of SATD. More specifically, we investigate the following research questions:

RQ1: How prevalent is SATD in smart contracts? We employed the Jitterbug tool (Yu et al. 2020) to detect SATD in relevant (real-world, non-toy) smart contracts. Next, we analyze the activity level (number of received transactions) and currency influx (total inbound USD) of contracts with SATD.

RQ2: How does the prevalence of SATD in smart contracts relate to code cloning? Recent studies have indicated that smart contracts are cloned at significantly higher rates than those observed in traditional software (Kiffer et al. 2018; He et al. 2019; Kondo et al. 2020; Gao et al. 2020). This makes code cloning a strong confounding factor in any analysis of smart contracts. In this research question, we investigate SATD prevalence and its diffusion in light of code cloning.

RQ3: What types of SATD do smart contracts contain? In this research question, we aim to gain a deeper understanding of the types of sub-optimal solutions that developers implement and acknowledge. Such an understanding can provide insight into techniques and tools to better support contract developers. To this end, we manually analyzed a statistically representative sample of SATD instances (comments) using open coding and developed a taxonomy of the different types of SATD.

Contributions. To the best of our knowledge, this is the first in-depth, large-scale study of SATD in Ethereum smart contracts. Our key contributions are as follows:

(i) describing what SATD means in the special context of smart contracts (Section 4.2); **(ii)** observing that SATD occurs in real-world smart contracts (e.g., CryptoKitties and OpenSea); **(iii)** relating SATD prevalence to code cloning, with the latter being performed not only by humans but also by other contracts; **(iv)** designing a taxonomy for SATD based on a thorough qualitative study (e.g., we observe SATD types that are specific to the context of smart contracts); and **(v)** providing a discussion of open research challenges and practical implications of our results (Section 4.7). A supplementary package is available here.

Chapter structure. Section 4.2 discusses the relevance of SATD in smart contracts, with a particular focus on how contract immutability impacts SATD repayment. Section 4.3 outlines our data collection methodology. Sections 4.4, 4.5, and 4.6 present the motivation, approach, and findings for each of our research questions. Section 4.7 examines the implications of our findings and identifies opportunities for future research. Section 4.8 discusses the threats to the validity of this chapter. Finally, Section 4.9 concludes the chapter.

4.2 The Role of SATD in Smart Contracts

4.2.1 Immutability & Contract Evolution

One of the fundamental characteristics of smart contracts is their immutability. Once deployed to Ethereum, the contract code cannot be modified or patched. Consequently, the evolution of a decentralized application (DApp) often entails deploying a new contract and migrating interactions to it, a practice commonly described as contract evolution. There are various mechanisms to facilitate this evolution

(Section 2.14). For instance, it can be performed manually (e.g., by implementing and invoking the `selfdestruct` function (Chen et al. 2021a)) or automatically (e.g., through the use of proxy pattern (Chen et al. 2021c)). However, automated upgrades are generally only possible if a contract evolution mechanism is integrated into the DApp's architecture from the outset. Such DApps often require a more complex design, which can result in increased testing and auditing efforts (Chen et al. 2021c).

From a governance perspective, contract evolution presents several challenges. Two key examples include: (i) how should end-users be migrated from one contract to another? for instance, should upgrades be enforced automatically or decided through a decentralized voting mechanism? and (ii) can end-users continue to trust a DApp if any smart contract can be seamlessly replaced at any time, potentially altering how funds or tokens are managed? Currently, there are no established guidelines for governing contract evolutions in DApps.

4.2.2 What Does Technical Debt Mean for Smart Contracts?

Managing technical debt is a crucial aspect of the software development process (Alves et al. 2016; Kruchten et al. 2019). It involves identifying suboptimal solutions (i.e., debt items) and performing code changes to improve or replace them. In conventional software systems, such code modifications are routine and integral to ongoing development. Once debt items are identified, addressing them typically involves (i) prioritizing them relative to other development tasks (e.g., implementing new features) and (ii) devising and applying an appropriate solution (i.e., a patch).

In the context of smart contracts, however, code changes are not straightforwardly applicable. Instead, contract evolution serve as the primary mechanism for modifying deployed logic. As discussed in the previous section, these upgrades are far more delicate than traditional code changes, due to the technical constraints and governance challenges involved. Practically speaking, this makes SATD significantly harder to resolve in smart contracts. If a contract do not interoperate an evolution mechanism, whether due to architectural choices or governance policies, or if the debt item is not deemed critical enough to justify an upgrade, the technical debt may become a permanent limitation of the contract. In other words, SATD in smart contracts is more likely to persist, potentially causing long-term issues for both contract authors and users (e.g., end-users or downstream developers).

As an initial step in this area, our study aims to examine the prevalence and nature of SATD in smart contracts. By analyzing these aspects, we seek to better understand the significance of SATD in this domain and to explore how it compares with technical debt found in traditional software systems.

4.2.3 Illustrative Examples of SATD in Smart Contracts

Example 1: Missing documentation. Listing 4.1 illustrates a smart contract containing a function that lacks documentation, with the presence of SATD indicated by a `TODO` comment. The contract in question implements a multi-signature (or Multi-Sig) wallet[1], which is a type of cryptocurrency wallet requiring confirmation from multiple signatories before executing fund transfers. Given the complexity of this

[1] https://etherscan.io/address/0xebacd5907d96b80e7d0f94af0292e4e50742d813

```
1  [...]
2  contract MultiSig {
3    [...]
4    // EVENTS
5    [...]
6    // FUNCTIONS
7    // TODO: document
8    function changeOwner(address _from, address _to) external;
9    function execute(address _to, uint _value, bytes _data) external returns (bytes32);
10   function confirm(bytes32 _h) returns (bool);
11 }
12 [...]
```

Listing 4.1: Missing source code documentation.

contract, comprising 1,311 lines of code and 51 functions, the absence of documentation introduces the risk of misunderstandings regarding its functionality. This is especially problematic because the contract directly handles financial transactions, and any confusion could lead to costly errors. As noted in a GitHub pull request[2], "*As this MultiSig can and is used in practice for very large funds, I thought it could be a good idea to enhance its documentation to better reflect how to use it and some common pitfalls.*"

Example 2: Workaround for a Solidity limitation. Listing 4.2 shows our second example involving a contract function[3] that implements a workaround to retrieve the return value of a call to the built-in Solidity function `ecrecover`. The underlying issue is that Solidity's inline assembly does not permit direct access to the return value of external function calls, making it challenging to safely utilize precompiled contracts such as `ecrecover`, which are commonly used for signature verification.

[2] https://github.com/ethereum/dapp-bin/pull/73
[3] https://etherscan.io/address/0xc78bb6c8848abe10c3a398fa1801926215d27f6f

To circumvent this limitation, the developer manually prepares memory to hold the necessary input parameters, performs a low-level call to `ecrecover`, and then reads the return value directly from memory. This technique cleverly reuses memory to indirectly capture the result, effectively working around the constraints of inline assembly, which lacks native support for handling return values from such calls.

```
1  /*
2  The following function has been written by Alex Beregszaszi (@axic),
3  use it under the terms of the MIT license Duplicate Solidity's
4  ecrecover, but catching the CALL return value
5  */
6  function safer_ecrecover(bytes32 _hash, uint8 _v, bytes32 _r, bytes32 _s) internal
       returns (bool _success, address _recoveredAddress) {
7    /*
8    We do our own memory management here. Solidity uses memory offset
9    0x40 to store the current end of memory. We write past it (as
10   writes are memory extensions), but don't update the offset so
11   Solidity will reuse it. The memory used here is only needed for
12   this context.
13   FIXME: inline assembly can't access return values
14   */
15   bool ret;
16   address addr;
17   assembly {
18      let size := mload(0x40)
19      mstore(size, _hash)
20      mstore(add(size, 32), _v)
21      mstore(add(size, 64), _r)
22      mstore(add(size, 96), _s)
23      ret := call(3000, 1, 0, size, 128, size, 32) // NOTE: we can reuse the request memory
            because we deal with the return code.
24      addr := mload(size)
25   }
26   return (ret, addr);
27 }
```

Listing 4.2: Workaround for a Solidity limitation.

4.3 Data Collection

Our study entails collecting the source code of smart contracts and detecting SATD in it. Figure 4.1 provides an overview of our data collection process.

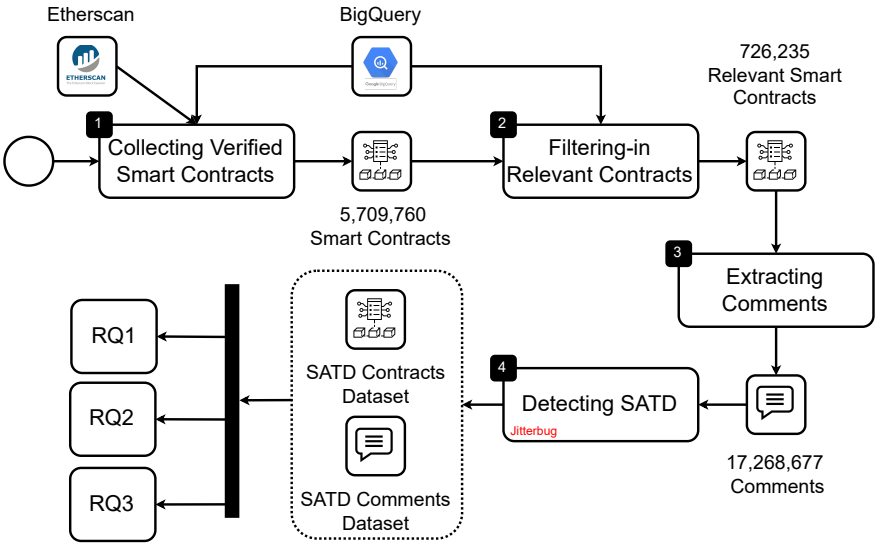


Figure 4.1: An overview of our data collection process.

Step-1: Collecting smart contracts. As the input of the process, we require a dataset of smart contracts. The dataset used in this work is a collection of all verified Solidity contracts hosted on Etherscan as of March 18th, 2019, which encompasses over 5.7 million contracts (5,709,760 to be exact). To collect verified contracts from Etherscan, we used the Ethereum dataset[4] from the Google BigQuery platform and obtained the addresses of *all* deployed contracts by means of an SQL query. Next, we used Etherscan's REST API to retrieve the contracts' source code. More specifically, for each address, we first checked whether the contract's source code is

[4]Dataset name: `bigquery_public_data.crypto_ethereum`

available and written in Solidity. In the positive case, we retrieved the source code (otherwise we skipped it).

Step-2: Filtering-in relevant contracts. We use a heuristic to filter-in real-world, non-toy contracts (henceforth referred to as *relevant contracts*). We define a smart contract as relevant if it satisfies either of the following criteria:

- Activity level: the contract received at least C_1 transactions
- Currency influx: the contract received at least C_2 USD worth of ETH.

To set C_1, we first determined the number of successful inbound transactions for each of the 41,769,738 contracts that had been deployed to Ethereum as of May 25th, 2021. We use this date to increase the likelihood of contracts to be seen as relevant, especially those created close to March 2019 (see Step 1). To do so, we once again leveraged the Ethereum dataset from Google BigQuery. Next, we made C_1 correspond to the 99th percentile of the distribution, which resulted in $C_1 = 29$ (i.e., 1% of all contracts in Ethereum received 29 or more transactions).

As for the second criterion, we set $C_2 = 100$ USD as the minimum deposit that a contract must have received (currency influx) to be considered relevant. Our rationale is that contract users would refrain from spending 100 USD just to play or experiment with a toy contract. To compute the USD currency influx of a contract, we first determined a daily ETH-to-USD exchange rate by averaging ETH's opening and closing prices from August 7th, 2015 (date of first contract transaction on Ethereum) to May 25th, 2021. The daily opening and closing prices where directly obtained from the Yahoo Finance[5] website. Finally, we used the Ethereum dataset from Google BigQuery once more to select transactions that sent ETH to contracts.

[5] https://finance.yahoo.com/quote/ETH-USD/history?p=ETH-USD

Finally, we converted the ETH amount to USD using the aforementioned exchange rates.

Having applied our criteria to all contracts, we identified 726,235 (12.72%) relevant contracts. More specifically, 123,259 (16.97% of all relevant contracts, 2.16% of all contracts) and 649,607 (89.44% of all relevant contracts, 11.38% of all contracts) are included via the first and second criterion respectively. Only 46,631 (6.41% of all relevant contracts, 0.82% of all contracts) are included by satisfying both criteria. Such a result highlights the complementary nature of the two criteria.

Step-3: Extracting comments. Once we obtained our set of relevant contracts, we then extracted their source code comments. We implemented an ANTLR parser using a popular Solidity grammar specification[6]. A total of 17,268,677 comments were extracted from our relevant contracts.

Step-4: Detecting SATD. We employed Jitterbug's *easy-to-find* component to detect SATD in relevant non-toy smart contracts. To do so, we trained the *easy-to-find* component on the refined SATD corpus by Yu et al. (2020), which builds upon the original dataset curated by Maldonado et al. (2017a). Table 4.1 describes the dataset in terms of the selected software systems, the domain of each system, and the number of comments as well as SATD comments. This corpus contains 62,275 comments and 4,071 SATD comments of the ten open-source software projects. This resulted in four SATD indicator patterns: `TODO`, `FIXME`, `HACK`, and `WORKAROUND`. We then searched for these indicator patterns in our extracted Solidity comments dataset, counting each match as an SATD instance. Contracts with at least one

[6] https://github.com/antlr/grammars-v4/blob/master/solidity/Solidity.g4

Table 4.1: Summary of the dataset used to train Jitterbug's easy-to-find component.

Project	Domain	Comments	SATD (Ratio)
JMeter	Testing	8,057	374 (4.64%)
Columba	Email Client	6,468	204 (3.15%)
Hibernate	Object Mapping Tool	2,968	472 (15.90%)
JFreeChart	Java Framework	4,408	209 (4.74%)
SQuirrel	Database	7,215	286 (3.96%)
Apache Ant	Automating Build	4,098	131 (3.20%)
ArgoUML	UML Diagram	9,452	1,413 (14.95%)
EMF	Model Framework	4,390	104 (2.37%)
JEdit	Java Text Editor	10,322	256 (2.48%)
JRuby	Ruby for Java	4,897	622 (12.70%)
Total	–	**62,275**	**4,071 (6.54%)**

SATD instance were labeled as *SATD contracts*, while those without were classified as *non-SATD contracts*.

Summary

Data Collection

- Smart contracts collection period: August 2015 to March 2019 (43 months).
- Data sources: Etherscan, Google Big Query
- Pieces of collected data: Source code of smart contracts, metadata associated with smart contracts (as of May 25th, 2021)
- Number of studied smart contracts: 5,709,760
- SATD detector: Jitterbug (easy-to-find) (Yu et al. 2020).

4.4 RQ1: How Prevalent Is SATD in Smart Contracts?

4.4.1 Motivation

Technical debt negatively affects various aspects of a software system, including software quality, evolvability (i.e., how quickly new functionality can be added), and maintainability (i.e., how effectively bugs can be prevented or resolved) (Cunningham 1992; Guo et al. 2011; Nugroho et al. 2011). While SATD has been widely studied in traditional software systems (Sutoyo and Capiluppi 2024), its prevalence in smart contracts remains underexplored (Bosu et al. 2018; Chen et al. 2021c). Unlike conventional applications, smart contracts are deployed on immutable ledger, making it significantly harder to address technical debt post-deployment. This fundamental constraint raises a critical question: do developers still self-admit technical debt when future refactoring is difficult or impossible? Given the lack of tooling and practices tailored to managing SATD in blockchain environments, understanding its prevalence is a necessary first step toward adapting or developing effective solutions. This research question seeks to fill that gap by quantifying how commonly SATD appears in Ethereum smart contracts.

4.4.2 Approach

We first determine the proportion of contracts with SATD. In addition, we characterize SATD contracts by analyzing their activity level and currency influx. We compare data distributions using parametric statistical tests and the Cliff's Delta (δ) measure of effect size. We use the following thresholds to interpret d (Romano et al. 2006):

negligible for $|\delta| \leq 0.147$, *small* for $0.147 < |\delta| \leq 0.33$, *medium* for $0.33 < |\delta| \leq 0.474$, and *large* otherwise.

4.4.3 Findings

Observation 4.1) ***1.5% of contracts contain SATD.*** We detected SATD in 10,549 relevant contracts, representing approximately 1.5% of all relevant contracts analyzed (726,235 in total). This prevalence, though not ubiquitous, is consequential because it demonstrates that developers knowingly deploy contracts with technical flaws. This practice is particularly alarming given that these contracts are immutable and often control substantial assets, making their admitted debt a permanent security and maintenance liability.

Observation 4.2) ***SATD occurs in extremely relevant smart contracts.*** The left-hand side of Figure 4.2 illustrates the activity levels of SATD-containing contracts. The distribution is highly right-skewed, with a pronounced peak in the 100 to 1,000 transaction range. A long tail extends beyond the 95th percentile (indicated by the red dotted line), suggesting the existence of a small subset of contracts that are exceptionally active in practice. Notably, the top five most active contracts received between 5.6 million and 33.6 million transactions. These include: FiatTokenProxy, a proxy contract used in the deployment of USD Coin (a stablecoin pegged to the U.S. dollar); KittyCore, the core contract of CryptoKitties, one of the most popular NFT-based games on Ethereum; CardMigration, a fundamental component of the Gods Unchained decentralized trading card game; GasToken2, a utility contract that enables users to store gas when it is cheap and redeem it later when gas prices are

high; and AdminUpgradeabilityProxy, a contract that implements the proxy pattern evolution mechanism with access control for administrative tasks.

The right-hand side of Figure 4.2 depicts the currency influx of SATD contracts. Similarly to the activity level curve, the currency influx curve also exhibits a long tail after the 95th percentile (albeit shorter). We thus conclude that there exists an additional small set of contracts that are also extremely relevant in practice (from a currency influx perspective). For reference, the top five richer contracts received between 84.3M and 352.2M USD worth of Ether. They are: Wyvern Exchange (a decentralized exchange for trading any kind of non-fungible assets, which is utilized by OpenSea, the largest NFT marketplace); the next top three contracts are copies of the MultiSig Wallet contract; and Etheroll (a smart contract for placing bets on a provably fair dice game using Ether).

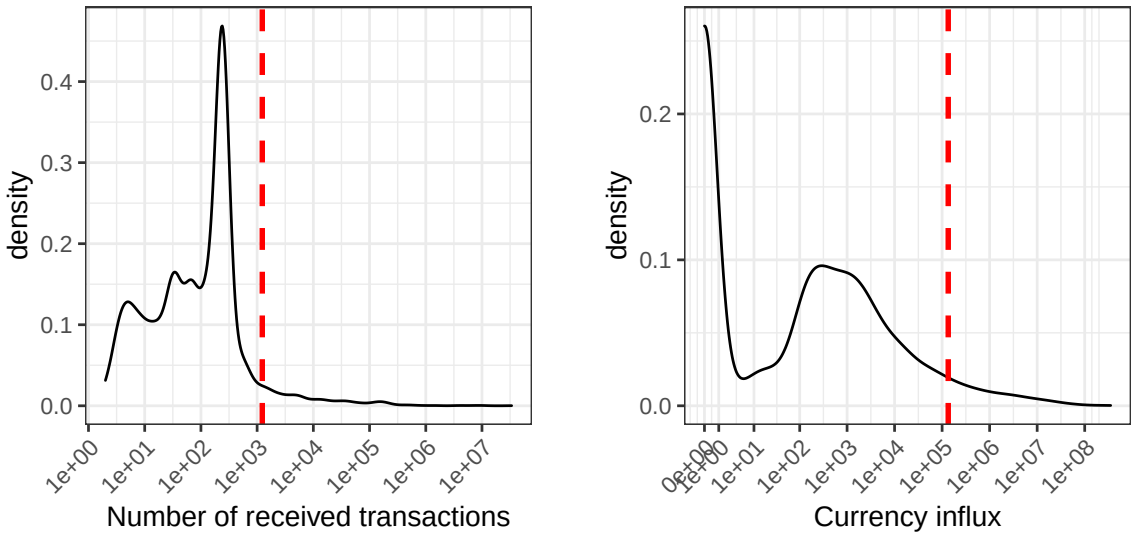


Figure 4.2: Distributions of activity level (left) and currency influx (right). Red lines denote the 95th percentile.

Observation 4.3) ***SATD contracts have a higher activity level than those without SATD.*** We observe that the median number of received transactions for SATD contracts is 118, which is approximately 17 times higher than that of non-SATD contracts. A two-tailed Mann–Whitney test ($\alpha = 0.05$) confirms that the difference between the distributions is statistically significant (p-value ≤ 0.05). Moreover, Cliff's Delta indicates that the effect size is large ($\delta = 0.6$). Hence, SATD contracts are used more frequently than those without SATD. This observation further emphasizes the practical relevance of SATD-bearing contracts.

Summary

RQ1: How Prevalent Is SATD in Smart Contracts?

SATD occurs in ∼1.5% of all relevant contracts. That is, some contracts are being put in production in spite of them having technical debt. In particular:

- SATD occurs in contracts that have received millions of transactions and/or millions of USD.
- SATD occurs in contracts associated with popular DApps, such as CryptoKitties (most popular NFT game) and OpenSea (largest NFT marketplace).
- Contracts with SATD typically received more transactions than those without SATD.

4.5 RQ2: How Does the Prevalence of SATD in Smart Contracts Relate to Code Cloning?

4.5.1 Motivation

Code cloning has been found to be exceptionally widespread in the Ethereum ecosystem. Early studies report that over 79% of contracts contain at least one duplicated

fragment (type-1 or type-2 clones) (Kondo et al. 2020), and more recent large-scale analysis suggests clone prevalence could reach 90% at various levels of granularity (file or function) (He et al. 2019). Cloning not only replicates functionality, it also propagates bugs, vulnerabilities, and poor design choices. For instance, He et al. (2019) observed that roughly 9.7% of similar contract pairs share identical vulnerabilities, suggesting that one buggy copy can lead to many fragile instances.

Taken together, these findings motivate us to investigate whether SATD commonly resides in cloned smart contracts and to what extent code cloning drives the prevalence of technical debt in Ethereum smart contracts. Understanding this relationship can help us pinpoint whether SATD is isolated or systemic, appearing in unique contracts or propagating through clone networks, and thus guide efforts to detect, manage, or prevent technical debt more effectively.

4.5.2 Approach

To investigate the relationship between SATD and code cloning, we performed the following analyses.

First, we study the the prevalence of cloning among contracts with SATD. We determine the number of contract clones, clone clusters, and clones per cluster among SATD contracts. To obtain those numbers, we employ the approach proposed by Kondo et al. (2020). First, we use the Cloc[7] tool to strip blank lines and comments out of our collection of SATD contracts. Next, we apply the *Deckard* (Jiang et al. 2007) clone detector. We chose Deckard because it is one of the few well-tested clone detection tools that supports the Solidity language. Since Deckard operates

[7] https://github.com/AlDanial/cloc

on Abstract Syntax Trees (ASTs), we need to ensure that it can correctly parse all the preprocessed contracts. After investigating Deckard's log files, we identified 210 (2%) contracts that Deckard failed to parse. We excluded these incompatible contracts from this analysis. As for Deckard's configuration, we set its similarity threshold to 0.79 based on the sensitivity analysis performed by Kondo et al. (2020) and leave other settings at their default values. In terms of output, Deckard detects clone pairs at the *code fragment level* (line ranges). Since our goal is to determine whether contracts with SATDs are clones of each other, we post-process these fragment level clone pairs to obtain file-level clones. We refer to a set of file-level clones as a *clone cluster*. Thus, we consider a contract to be a *clone* if its cluster is *not* a singleton (i.e., the size of its cluster is higher than or equal to 2).

Second, we study the evolution and deployment authorship of cloning. To have a grasp on how cloning numbers evolve and who deploys clones (deployment authorship), we study the top-10 largest clone clusters. To study cloning evolution, we sort contracts within each clone cluster in ascending order of deployment timestamp. Next, we analyze these deployment timestamps to uncover trends in the evolution of cloning activities. With regards to deployment authorship, our goal is to determine who is behind the cloning activities associated with each cluster. To this end, we study the percentage of clones created by externally-owned accounts (EOA) versus contract accounts (CA) for each cluster. Finally, we also determine whether clones inside a cluster tend to be deployed by the same deployer account. To this end, we compute the Gini index for the *number of contract deployments per account*. The Gini index is a measure of inequality (Ceriani and Verme 2012). A Gini index of 0% indicates lack of inequality (e.g., when each account deploys the

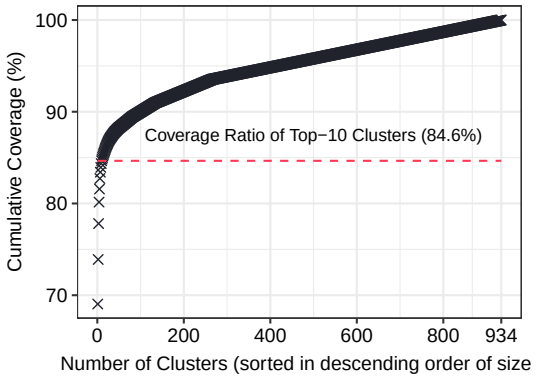


Figure 4.3: Cumulative coverage of clone clusters in terms of the ratio of SATD contracts.

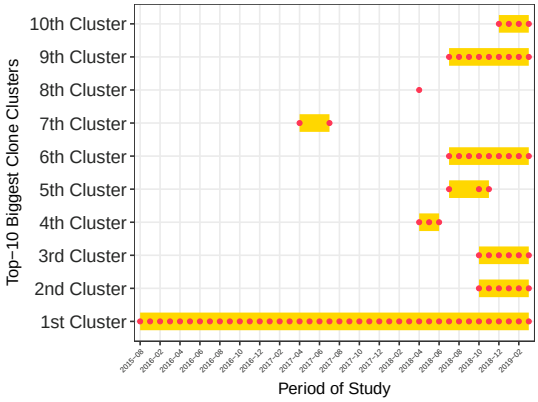


Figure 4.4: Evolution of the top-10 largest clone clusters.

exact same number of contracts). A Gini index of 100% indicates the opposite (e.g., when a single account deploys all the contracts).

4.5.3 Findings

Observation 4.4) ***93.5% (9,671) of SATD contracts are clones.*** In other words, only 6.5% (668) of SATD contracts do not have a cloned version.

Observation 4.5) ***The top-10 clone clusters contain together 84.6% of all SATD contracts.*** We identified 934 clone clusters (including singleton ones). Figure 4.3 depicts the relationship between the *number of clusters* and the *percentage of covered SATD contracts*. The red dashed line in the graph indicates the proportion of SATD contracts that belong to the top-10 clusters. Such a proportion is 84.6%, which indicates that very few clusters concentrate the vast majority of SATD contracts. In particular, the largest cluster contains 69% of all SATD contracts.

Given the considerable share of the largest clone cluster, we analyze it in more detail. First, we study how homogeneous this cluster is by determining the share of the largest group of identical contracts (a.k.a., type-1 clones) contained therein. More specifically, we compute the `md5` hash of all contracts in the cluster then group those contracts with identical hash. We observe that the largest group of type-1 clones encompasses 99.9% of all contracts in that cluster. That is, almost all contracts within this cluster are identical.

A manual inspection of this heavily cloned contract[8] (MultiSig) reveals that it implements a traditional multi-signature wallet (Listing 4.1). The source code was authored by Gavin Wood, a co-founder of Ethereum. Further analysis indicates that this contract was a core component of the now-deprecated Ethereum Wallet application[9], developed by the Ethereum Foundation. When an end-user initiated the creation of a multi-signature wallet through the Ethereum Wallet interface, the application compiled the MultiSig contract source code and deployed a new instance on the blockchain using the end-user's Externally Owned Account (EOA). This deployment model resulted in numerous byte-for-byte identical contract instances, each corresponding to an individual user's request. In Appendix A.4, we describe the other nine clone clusters in more detail.

Observation 4.6) ***Most SATD contracts have been systematically cloned over time.*** Figure 4.4 depicts the evolution of the top-10 largest clone clusters. A red dot indicates that at least one contract was created in a given month for a particular clone cluster. Each gold line connects the left-most red dot to the right-most red

[8] https://etherscan.io/address/0x2ebc5330ec52107a654009d0e250f385d969c159
[9] https://github.com/ethereum/mist

dot, denoting the duration of cloning for a given clone cluster. Analysis of Figure 4.4 led us to distinguish between three types of cloning practices:

- Punctual cloning (single month): cluster 8
- Short-term cloning: cluster 4, 5 and 7
- Ongoing cloning: all other clusters. Cluster 1 is clearly systematic (cloning happening almost every month). Clusters 2, 3, 6, 9, and 10 also appear to be ongoing.

The diffusion trends show that for many clusters cloning occurs across long time spans, sometimes persisting for months. This means that SATD is not just introduced once and forgotten. Instead, it is continually propagated, often as a byproduct of copy-pasting. The systematic nature calls for better lifecycle management, shared libraries with proper versioning, and developer education to reduce the silent spread of technical debt.

Observation 4.7) ***The majority of clones are automatically deployed by other contracts and tools rather than human developers.*** Table 4.2 displays information regarding deployment authorship per contract cluster. In 9 out of 10 clone clusters, a single type of account is responsible for at least 99% of all deployments. That is, cloning activity in a cluster can almost always be attributed exclusively to either EOAs or CAs.

More specifically, EOAs are responsible for the cloning activity happening in four clusters (clusters 1, 5, 7, and 10). In turn, CAs are responsible for cloning activity in five clusters (clusters 2, 3, 4, 6, and 8). In Cluster 9, approximately 70% of the clones are deployed by EOAs. In summary, five clone clusters exist due to CAs. Also, as we discussed before, the contracts in cluster 1 are likely deployed by a tool

Table 4.2: Contract deployments by account type in top-10 clusters.

Top-10 Clusters	Deployer's Account Type					
	External-owned accounts (EOA)			*Contract accounts (CA)*		
	Contracts Deployed (%)	*Unique Deployers*	*Gini Index for Deployments per Deployer*	*Contracts Deployed (%)*	*Unique Deployers*	*Gini Index for Deployment per Deployer*
Cluster 1	**7,138 (100%)**	**5,323**	**0.24**	0 (0.00%)	0	NA
Cluster 2	0 (0.00%)	0	NA	**502 (100%)**	**404**	**0.18**
Cluster 3	0 (0.00%)	0	NA	**405 (100%)**	**1**	**1.00**
Cluster 4	0 (0.00%)	0	NA	**242 (100%)**	**8**	**0.42**
Cluster 5	**147 (100%)**	**2**	**0.49**	0 (0.00%)	0	NA
Cluster 6	1 (0.84%)	1	1.00	**118 (99.16%)**	**1**	**1.00**
Cluster 7	**71 (100%)**	**5**	**0.72**	0 (0.00%)	0	NA
Cluster 8	0 (0.00%)	0	NA	**54 (100%)**	**2**	**0.44**
Cluster 9	**43 (69.77%)**	**17**	**0.32**	13 (30.23%)	8	0.26
Cluster 10	**31 (100%)**	**1**	**1.00**	0 (0.00%)	0	NA

and not humans (even though the deployer is registered as an EOA). Therefore, we conclude that the vast majority of clones exist due to cloning performed by other contracts and tools (instead of humans).

Finally, we explored whether a single account is responsible for the majority of cloning activities. The results of Gini index are detailed in columns 4 and 7 of Table 4.2. In summary, the Gini index is greater than zero in all the clones clusters, which indicates that there exists at least one account which has deployed multiple clones of a SATD contract in every cluster. In particular, in clusters 3, 6 and 10, a single account deployed all the SATD contracts.

4.5.4 Code Cloning as a Confounding Factor

The observations highlighted so far suggest that SATD is strongly associated with code cloning. In this sense, cloning acts as a major confounding factor when studying SATD-bearing contracts. In this subsection, we aim to better understand SATD contracts by *explicitly* accounting for code cloning.

4.5.4.1 Approach

We conduct conduct three focused analyses. First, we analyze the number of SATD instances per contract. To mitigate the impact of code clones, we select a single *representative contract* from each clone cluster, including singleton clusters. The representative is chosen as the contract with the highest number of type-1 clones within its cluster. We then report descriptive statistics for the variable *number of SATD instances per representative contract*. We choose this correlation coefficient in lieu of others because it does not require the data to be normally distributed.

Second, we study the association between the number of SATD instances and contract size. We investigate the extent to which the size of SATD contracts is associated with the number of SATD instances that appear in those contracts. First, we select representatives using the aforementioned approach. Next, we compute contract size using lines of code (LOC) as given by the Cloc tool (we note that blank lines and comments do not count towards LOC). Then, we calculate the *Spearman's correlation coefficient* between LOC and the number of SATD instances. We choose this correlation coefficient in lieu of others because it does not require the data to be normally distributed.

Third, we analyze common practices in self-admitting technical debt in smart contracts. We examine the SATD indicator patterns (e.g., `//TODO`) that developers use to document technical debt, based on the out-of-the-box patterns provided by Jitterbug (Section 4.3). Additionally, we study the commenting styles (e.g., single-line comments using `//`) used in conjunction with these indicators. For this analysis, our input consists of all textually unique SATD instances identified in the dataset.

4.5.4.2 Findings

Observation 4.8) ***72.4% of representative SATD contracts has a single SATD item.*** In particular, the median and standard deviation of the number of SATD items per representative contract are 1 and 1.45 in turn.

Observation 4.9) ***There is no association between the SATD count and contract size.*** The Spearman coefficient is 0.02, indicating no meaningful relationship, consistent with prior findings in traditional software Potdar and Shihab 2014. Figure 4.5 shows that most SATD contracts contain a single instance, regardless of their size.

Observation 4.10) ***Developers mainly employ the*** `//TODO` ***pattern in a simple single-line comment (//) to document SATD.*** We observe 555 textually-unique SATD instances. The vast majority of those employ the `todo` pattern. More specifically, `todo` accounts for 494 (89.01%) of all SATD instances, `hack` accounts for 27 (4.87%), `workaround` accounts for 25 (4.5%), and `fixme` accounts for only 9 (1.62%). Finally, we identified that developers employ four different styles of comments to express SATD. Simple single-line (`//`) is the most prevalent style, accounting for 474 (85.41%) of all SATD instances. NatSpec multi-line style (`/** **/`) accounts for 54 (9.73%), simple multi-line accounts (`/* */`) accounts for 18 (3.24%), and NatSpec single-line (`///`) accounts for only 9 (1.62%). The most common combination of pattern and style is `TODO` with a simple single line, accounting for 447 (80.54%) of all SATD instances.

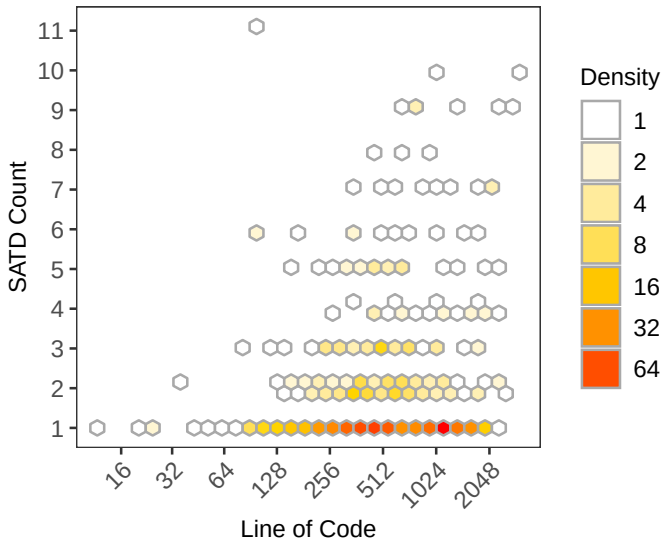


Figure 4.5: Correlation heatmap between SATD count and contract size.

Summary

RQ2: How Does the Prevalence of SATD in Smart Contracts Relate to Code Cloning?

SATD prevalence is strongly associated with code cloning, as 93.5% of all SATD contracts are clones. In particular:

- The top-10 clone clusters contain together 84.6% of all SATD contracts.
- Most SATD contracts have been systematically cloned over time.
- The majority of clone clusters exist due to cloning performed by other contracts and tools (instead of humans).
- 72.4% of SATD contracts contain a single SATD instance (controlled for clones)
- 80.54% of all SATD instances are expressed with `//TODO` (controlled for clones)

4.6 RQ3: What Types of SATD Do Smart Contracts Contain?

4.6.1 Motivation

While prior work has extensively categorized SATD types in conventional software systems (Alves et al. 2014; Alves et al. 2016; Rios et al. 2018; S. Maldonado and Shihab 2015), little is known about how these categories manifest in blockchain-based systems. The Ethereum ecosystem introduces unique design constraints, such as gas costs, immutability, and limited storage, which may give rise to debt types not commonly seen in traditional settings. Given the large volume of existing SATD classification studies in mature domains like enterprise, open source, cloud-based software, we are motivated to explore whether smart contracts reflect similar categories or exhibit domain-specific patterns of SATD. Understanding these patterns can inform better tooling and documentation practices for developers and offer contract consumers greater transparency into potential limitations or risks, especially when certain debt types recur across projects.

4.6.2 Approach

Similar to Xiao et al. (2022), we applied a mixed method that consists of combining open coding and card sorting techniques to build a taxonomy of SATD types. The input to this process is a dataset of 555 distinct SATD instances. We obtained such unique instances by removing textually duplicate SATD instances from our dataset. For each SATD instance, we randomly picked a smart contract that contains such

SATD to be investigated. Given this input, we planned our qualitative study methodology in four phases: i) Collaborative open coding, ii) Code structure finalization iii) Evaluation, iv) Taxonomy generation. In the following, we explain each of these phases in detail:

i) Collaborative open coding. In this phase, the first two authors (i.e., the coders) collaboratively perform open coding. The goal of this step is to extract a curated set of codes that represent the underlying problems behind SATD instances. To achieve such analytical generalization, we utilize the *theoretical saturation* method (Eisenhardt 1989). Using this technique, the coders incrementally code randomly picked SATD instances until no new codes emerge. Similar to prior studies (Hirao et al. 2019; Xiao et al. 2022), we set our saturation threshold to 50. This means that the coders stop the process if and only if no new code is generated upon coding 50 consecutive SATD instances.

Concerning the details of coding, we determined a protocol and strictly followed that for every SATD instance. The coders open the contract that contains a given SATD and rigorously read the context around the line of code where SATD was introduced. Whenever necessary, the coders use external resources (e.g., Solidity documentation, GitHub issue reports, StackExchange posts, etc.) to better the SATD and its context. For example, there were cases in which we needed to understand the Solidity features available in the version with which the contract was compiled. Next, the coders open a discussion to verify that the SATD indeed refers to a sub-optimal solution. Subsequently, if both coders agree on a code, they assign it to that SATD instance. According to the open coding protocol, if they cannot reuse

existing codes, they derive a new one and store it in a codebook[10]. This process may also result in revising old codes to keep them as mutually exclusive as possible. In situations when both coders are doubtful whether a comment is a SATD at all, they assign *Inconclusive*. If coders recognize that a comment is mistakenly classified as debt by the employed SATD detection tool, they assign *False Positive*. Finally, in case coders cannot reach an agreement for the coding of a given SATD instance, the third author intervenes and solves the disagreement.

We reached saturation after coding 237 SATD instances, which is slightly greater than the size of a statistically representative sample (228 SATD instances) with confidence level and interval set to 95% and 5 respectively. The collaborative coding process took 57 man-hours.

ii) Code structure finalization. Having reached a saturation point, the coders perform another pass to finalize their code structure. In addition, this pass ensures consistency because sometimes codes that are created late in the process may better represent early coded SATD instances. Having performed this pass, we slightly altered the codes of six SATD instances.

iii) Evaluation. The goal of this phase is to assess the reliability and generalizability of our coding structure. In particular, we first randomly pick another sample of 50 SATD instances from the remaining 318 SATD instances that were not coded during the first phase. Given the list of codes and the coding protocol from phase one, the coders now independently code this new sample. We also consider an extra code called *Other* for situations when a coder recognizes that none of the existing codes are appropriate for a SATD instance. This is followed by a *consensus*

[10]Available here: https://www.dropbox.com/sh/kykwepjuxso5ld0/AACqDlTX6yVS68X5jEjo-Z82a?dl=0

meeting where the coders reach a consensus on conflicting cases (12 out of 50). The code chosen by the second coder tended to prevail during the discussion of conflicting cases. This is due to his larger practical and academic experience in the domain of blockchain. All conflicts were solved during the meeting (the third author did not need to intervene). During the meeting, the coders also discuss the *Other* cases and improve the codebook together. To measure reliability, we compute the Cohen's Kappa coefficient to calculate the inter-coder agreement. The resulting coefficient is 0.71, which indicates a substantial level of agreement. This gives us high confidence about the quality of our codes. It is important to point out that we ended up creating only one new code after the consensus meeting, namely *lack of obfuscation*. To investigate the prevalence of this particular code, we searched for the word *obfuscation* in all remaining SATD instances and did not find any match.

iv) Taxonomy generation. We then perform card sorting on our extracted codes. Initially, the first author extracts a reference list of technical debt types and definitions from prior studies conducted in other domains (Alves et al. 2016; Bavota and Russo 2016; Rios et al. 2018; Liu et al. 2020). Then, the first two authors run a hybrid card-sorting meeting during which they organize and abstract codes into major groups and eventually SATD types. We use hybrid card sorting because of two reasons. Firstly, the literature presents solid empirical knowledge on types of SATD in domains other than smart contracts (Bavota and Russo 2016; Rios et al. 2018; Liu et al. 2020). Secondly, part of this knowledge is directly applicable to smart contract, since contracts are ultimately just code. Given this, hybrid card sorting enables us to incorporate this knowledge as a basis for grouping codes while giving us authority to generate new groups and SATD types if necessary.

4.6.3 Findings

Observation 4.11) ***We detected six major types of debt in the smart contract domain.*** Figure 4.6 shows our taxonomy of SATD types and the number of instances that belong to each category. Of 287 coded instances, 40 (13.9%) and 18 (6.3%) cases were false positive and inconclusive, respectively. We also had a single conflict that was resolved by the third author accordingly. In the following, we discuss SATD types along with their subcategories. In the following, we discuss each type. Examples are shown in blue.

1. Design debt refers to issues related to the design of a smart contract such as misplaced code, unclear design, non-standard design, or lack of abstraction/encapsulation. We found 16 (5.6%) instances of this SATD class. In the following, we discuss different subcategories.

• *Misplaced responsibility* refers to the issues and tradeoffs made with respect to where developers should locate a piece of code. Such SATDs often imply that a piece of code should be relocated to somewhere else (e.g., libraries) or potentially extracted into a different module. SATDs of this class often violate the single-responsibility principle and contribute to low cohesion, increasing the friction in software maintenance or changeability.

"*TODO: Move all insert validation into the DLL lib*"

• *Unclear design* refers to developers' uncertainty with respect to various design choices. Doubts regarding the degree of belongingness of a function to a contract, the visibility of variables, or proper identification of contracts to maximize modularity are some examples in this category. In the following example, developers

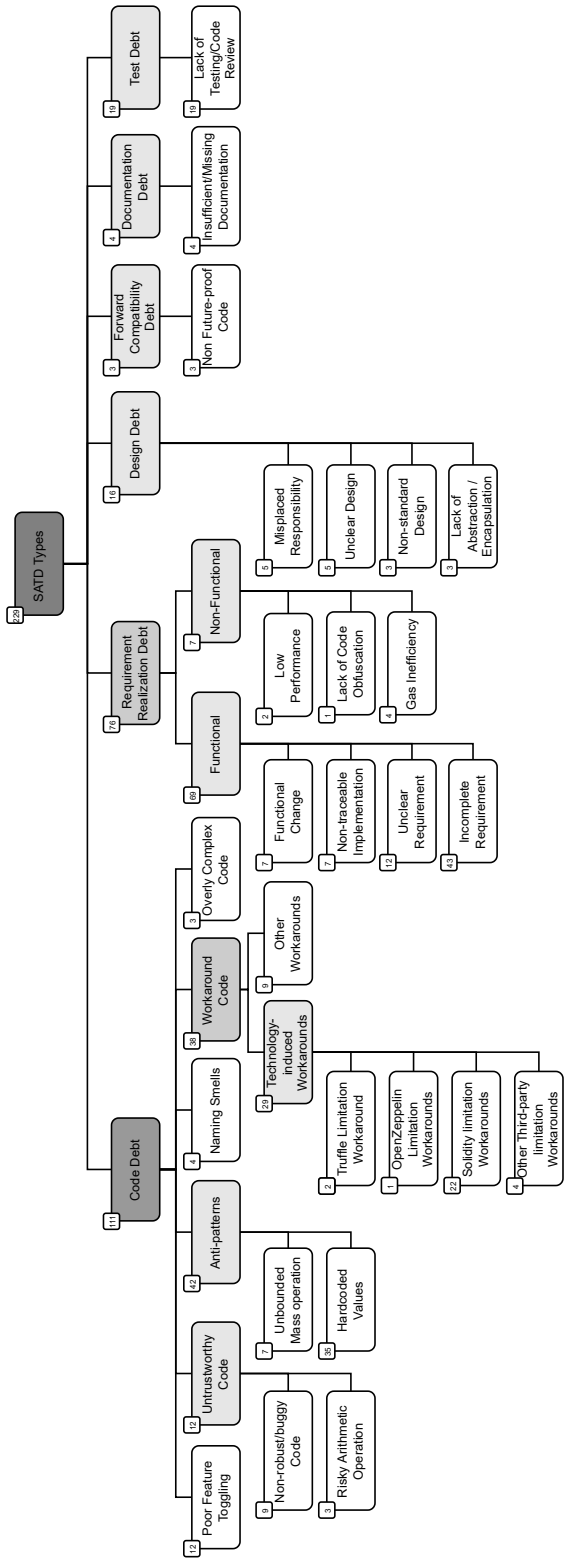


Figure 4.6: Our classification of SATD types found in the studied smart contracts. Each box shows a specific subtype with its number of occurrences.

are wondering if they need to create an interface contract rather than the existing solution:

"TODO: - create MonetarySupervisor interface and use it instead?"

• *Non-standard design* refers to issues concerning the lack of adoption of best practices or known standards. For instance, integer overflow and underflow are two disruptive common errors regarding arithmetic operations in solidity. That said, OpenZeppelin released a library called SafeMath that provides safe interfaces for performing such basic operations. In the following example, developers stated to employ the SafeMath contract which is nowadays a well-known accepted solution:

"todo use safemath.sol"

• *Lack of abstraction/encapsulation* refers to the lack of two important software engineering practices. The former employs abstraction to reduce duplication whereas the latter one hides the state of data objects inside a contract or method to avoid unauthorized access. The following two examples highlight the abstraction and encapsulation concerns in turn:

"TODO - you can get a way with a single generic verifier."

"TODO define accessors instead"

2. Documentation debt refers to issues related to improper source code documentation via commenting. We found 4 (1.4%) instances of this SATD class.

• *Insufficient/missing documentation* both refer to the lack of proper documentation at the code level. The first motivating example in Section 4.2 shows a debt of this class.

"TODO : document"

3. Test debt refers to issues related to the quality assurance (QA) of a smart contract. We found 19 (6.6%) instances of this SATD class.

• *Lack of testing or code reviews* refer to the necessity of conducting (more) unit testing or code review activities with regards to implemented functionalities. For instance:

"*TODO: complicated function test it thoroughly!*"

"*TODO need double check the authority checking*"

4. Code debt refers to an umbrella of poor coding practices (a.k.a., smells) such as overly complex codes, naming smells, anti-patterns, workarounds, untrustworthy codes. We found 111 (38.7%) instances of this SATD class.

• *Overly complex code* refers to a piece of code that could have been written in a shorter simplified way. In the following example, developers admitted that the current implementation is complex:

"*TODO there must be a shorter way to do this:*"

• *Naming smells* refer to issues related to bad naming practices such as non-representative names, spelling errors, etc. In the following example, developers stated that the chosen name does not represent what the function does. Such smells can negatively affect code readability and comprehensibility.

"*TODO: consider renaming operator depending on how this is implemented*"

• *Anti-patterns* refer to a class of recurrent bad practices in smart contract programming that are often inappropriate, ineffective and risky such as unbounded mass operations and hard coded values.

  - *An unbounded mass operation* (a.k.a., unbounded for-loop) (Tikhomirov et al. 2018; Grech et al. 2020; Kannengiesser et al. 2021) refers to issues that are caused by having a loop that has no constraint on its number of iterations. Such loops are generally classified as a weakness for smart contracts because they are prone to transaction failure or vulnerabilities. For instance:

    "todo Look into making a mass event instead of a one by one if Events use gas"

  - *A hard coded value* refers to embedding data (e.g., asset prices, addresses, etc.) directly into a smart contract's source code rather than obtaining it at runtime (e.g., via constructor parameters), making the smart contract a rigid non-reusable software.

    "TODO: change addressed before deployed to main net"

- *Workaround code* refers to a temporary, hacky, or shortcut solution that bypasses an existing limitation, bug, or problem. We divided this category based on the origin of the workaround into technology-induced workarounds and other workarounds.

  - *Technology-induced workarounds* refer to workaround solutions that are implemented due to limitations of or bugs in the development technologies such as programming languages (e.g., Solidity limitations), frameworks (e.g., Truffle limitations), libraries (e.g., OpenZeppelin limitations), and other third-party limitations.

    * Solidity limitation workarounds *"TODO Remove this library once solidity supports passing a signature to ecrecover. See `https://github.com/ethereum/solidity/issues/864`.* See also the second example shown in Section 4.2.

    - Truffle limitation workarounds: *"FIXME after fix of truffle issue #560: refactor to a separate contract file which uses InvestmentAnalytics interface"*
    - OpenZeppelin limitation workarounds: *"workaround since vault refund does not return refund value."*
    - Other third-party limitation workarounds: *"Hack to get things to work automatically on OpenSea."*

  – *Other* refers to other workarounds that are not technology-induced.

    "This is a workaround to surpass an issue that Mintabletoken functions are not accessible in this contract. We did not want to change the Openzeppelin code and we did not have the time for an extensive drill down."

• *Untrustworthy code* refers to implementations that are potentially non-robust, buggy, or error-prone.

  – *Non-robust/buggy code* refers to implementations that are either non-robust or defective. For instance, codes that cannot handle corner cases.

    "TODO: This will introduce an error for very large numbers because the error in our Pi will compound. But for actual reasonable angle values we should be fine."

  – *Unsafe arithmetic operation* refers to operations that are prone to overflow or underflow errors and potentially need further tests or refactoring measures.

    "TODO - check overflow"

• *Poor feature toggling* refers to a poor coding practice in enabling and disabling features via commenting and uncommenting pieces of codes. For instance:

"TODO: Remove this method before deploying Smart Contract."

5. Requirement realization debt refers to trade-offs related to what requirements should be implemented and how to implement them. We identified 76 (24.5%) instances of this class. We further divided this class into *functional* and *non-functional* subcategories.

• *Functional (requirements)* refers to the services that the contract must provide to its end-users. We observed the following issues:

– *Functional change* refers to situations where the behavior of a piece of code (e.g., function) needs to be changed. For example:

*"TODO: If ether gets accidentally trapped in this contract by some other transaction this function will send it back to the primary in the subsequent order. Change code to only return back what's left over from *this* transaction."*

– *Non-traceable implementation* refers to pieces of code that cannot be easily traced back to requirements, making them subject to removal. For example:

"TODO consider removing. Can this condition be met?"

– *Unclear requirements* refer to developers' uncertainty about requirements or their correctness. For instance, in the following example developers are not sure if they should implement the goalkeeper constraint.

"TODO: Validate that there's at least 1 goalkeeper?"

– *Incomplete requirements* refer to requirements that are missing or partially implemented and need to be completed. For example:

"todo: implement group add someday however func distributeAndReturnToPool gets very complicated"

• *Non-function (requirements)* refer to the quality attributes of a system (as opposed to its behavior). We observed the following issues:

  – *Low performance* refers to concerns about the low performance of a smart contract's functionality. For example:

    "*todo: change this to something faster*"

  – *Gas inefficiency* refers to concerns with respect to saving gas in smart contracts. This is because a low gas consumption is a critical non-functional requirement for almost any smart contract. Note that an unbounded loop operation can lead to gas-related issues. Nevertheless, we decided to consider it as a separate subcategory under poor code due to its relevance as an anti-pattern (Grech et al. 2020). For example:

    "*TODO optimize for gas usage*"

  – *Lack of obfuscation* refers to the lack of code obfuscation. Code obfuscation is a method for deliberately creating or modifying an executable source code to prevent others from understanding what it does. For instance:

    "*Constructor: ToDo: obfuscate*"

6. Non-future-proof code refers to situations where the existing implementation can be outdated or suboptimal in the future due to the evolution of surrounding technologies. In the following example, developers implemented a piece of code that checks if a given address refers to a *contract account* or an *externally owned account*. The SATD instance implies that after the Serenity release (a.k.a., Ethereum 2.0, which is planned to be released mid 2022), developers will need to recheck the

code because "all addresses will be contracts then". The SATD thus suggests that the existing code is not forward-compatible with Ethereum protocol upgrades:

"XXX Currently there is no better way to check if there is a contract in an address than to check the size of the code at that address. See `https://ethereum.stackexchange.com/a/14016/36603` for more details about how this works. `TODO` Check this again before the Serenity release, because all addresses will be contracts then."

Observation 4.12) ***Many of our low-level detected SATD categories (11) are unique to smart contracts.*** At the macro level, our detected SATD types align with five of the six major categories identified in prior work on conventional software Bavota and Russo 2016, with the addition of a new type, *forward compatibility* debt, emerging from the rapid evolution of blockchain technologies. Consistent with Bavota and Russo 2016, code debt (38.7%) and requirement realization debt (26.5%) are the most prevalent in smart contracts. While some micro-level categories (e.g., *lack of testing/code review*, *insufficient/missing documentation*) are common across software domains, we identify eleven types unique to smart contracts, including *poor feature toggling*, *risky arithmetic operations*, four types of *technology-induced workarounds* and two types of *anti-patterns*, *gas inefficiency*, *lack of code obfuscation*, and *non future-proof code*.

Observation 4.13) ***Contract developers sometimes forget to remove a SATD instance once the debt is paid off.*** For a given SATD instance, if we understand that it is already paid off, we categorized it as a false positive. We refer to these code comments as *outdated SATD comments*. Outdated SATD comments represent almost half (43.6%) of all false positives. Outdated SATD comments can mislead both contract authors and consumers (e.g., end-users, other developers reusing the

contract), as sometimes it is not trivial to determine whether a SATD comment is outdated. In our experience, this is particularly true when the piece of code that pays off the SATD is located far away from the SATD in the source code (e.g., more than a hundred lines away).

Summary

RQ3: What Types of SATD Do Smart Contracts Contain?

We identified six types of SATD: code debt (38.7%), requirement realization debt (26.5%), test debt (6.62%), design debt (5.6%), documentation debt (1.4%), and forward compatibility debt (1%). In particular:

- Although our major SATD types are similar to those observed in traditional software (Bavota and Russo 2016), several of our low-level SATD types are tied to the domain of smart contracts.
- Practitioners should pay attention to the removal of SATD instances once the debt is paid off to avoid misconception.

4.7 Discussion

4.7.1 On the Connection Between SATD and Cloning

From RQ2, we observe a strong connection between SATD and cloning. Such a connection raises several research questions that should be addressed by future research. In the following, we discuss two scenarios:

• **DApps that follow the "one deployment per end-user" paradigm and that contain a contract with SATD.** The *Ethereum Wallet DApp* discussed in RQ2 (cluster 1) is an example of a DApp that follows the aforementioned paradigm (Observation 5). In other words, every time a new user starts using the Ethereum Wallet DApp, a new contract with SATD is deployed. This scenario leaves us with several open questions, including: (q1) does releasing the DApp outweigh the inherent downsides of the SATD?, (q2) is the SATD a minor issue?, (q3) can the SATD contract be easily upgraded, thus rendering the SATD instance less problematic?

• **Frameworks and libraries that contain a contract with SATD.** An example can be seen in the clone cluster 3 mentioned in RQ2. More specifically, clone cluster 3 is composed of identical copies of a single contract called KernelProxy. KernelProxy is a component of AragonOS, which is a smart contract framework for developing robust (e.g., upgradable, permission-controlled) DApps on Ethereum (Aragon 2023). We thus wonder (q4) whether DApp developers are aware of SATD instances in contracts belonging to frameworks and libraries (dependencies). Assuming that they are aware of the SATD, then (q5) do they fully understand the consequences of the SATD? and (q6) do they use the SATD contract because no better alternative exists (or a better alternative is not known)?

Answering the open questions above (q1-q6) is important, since all the clones that we identified in our study are relevant, real-world contracts (Section 4.3). Yet, answering them is far from trivial. First, it would require designing novel methods for identifying key stakeholders (e.g., the developers behind a DApp that either produces or consumes SATD contracts), since Ethereum accounts do not contain

developers' contact information. Second, it would require designing and conducting an in-depth qualitative study (e.g., an online survey followed by interviews with DApp developers). We thus strongly encourage future research to take on this endeavor, which would shed more light into the interplay between SATD and cloning.

4.7.2 Implications

Implication 4.1) ***Certain types of SATDs are associated with coding anti-patterns, for which solutions exist.*** Anti-patterns are *recurrent* poor programming practices. Our study identified several SATD instances that correspond to well-established coding anti-patterns, notably *hard-coded values* and *unbounded mass operations* (Observation 11). Hard-coded values hinder contract reusability and adaptability, and may increase vulnerability risk when sensitive values, such as fixed addresses or thresholds, are embedded in the source code (Chen et al. 2022c). This issue can be resolved by passing such parameters through the contract's constructor or designated function inputs, allowing values to be configured at deployment or runtime without modifying the underlying code. Similarly, unbounded loops over dynamic data structures may cause transactions to run out of gas or exceed the block gas limit, leading to execution failures (Kannengiesser et al. 2021). This problem is particularly concerning given Ethereum's gas constraints and transaction fee model. We found different solutions to this anti-pattern, including shifting computational work to the user side (e.g., pull-based logic), resuming long loops via check pointing mechanisms, or partitioning execution into smaller batches processed across multiple transactions (Kannengiesser et al. 2021; Chittoda 2019). The recurring nature

of these debt patterns underscores the need for integrated development environments (IDEs) and frameworks to provide automated support for their detection and resolution. IDE plugins or static analysis tools tailored for smart contract platforms (e.g., Truffle, Hardhat, Foundry) should flag such SATD-indicative comments, offer context-aware warnings, and recommend corrective patterns.

Implication 4.2) ***In response to deprecations and the rapid advancement of the Ethereum ecosystem, developers should design DApps that support contract evolution.*** Despite the rapid growth of the Ethereum protocol and underlying technologies (e.g., Solidity language, frameworks, and platforms), developers cannot always keep up with demand. More specifically, certain SATDs clearly show gaps between the needs of contract developers and what Ethereum and its ecosystem has to offer. This is evident in the prevalence of *technology-induced workarounds*, where developers explicitly acknowledge having to circumvent limitations in Solidity, libraries like OpenZeppelin, or platforms such as Truffle. Conversely, some SATD reflects anticipation of upcoming changes to Ethereum that may render certain design choices obsolete. These instances of *non-future-proof* code suggest an awareness among developers that technical debt may stem not from negligence, but from the shifting foundations of the platform itself. The immutable nature of smart contracts heightens the significance of this issue, since post-deployment modifications are non-trivial, costly and risky. In this context, designing DApps with contract evolution in mind becomes not only advisable but essential. Developers are encouraged to incorporate evolution mechanisms early in the design process and to carefully evaluate available mechanisms, such as the proxy pattern or `CREATE2`, based on the

project's complexity, governance model, and security requirements. Established libraries, such as OpenZeppelin's upgradeable contract suite (OpenZeppelin 2023b), provide ready-to-use implementations of proxy pattern evolution that can reduce the barrier to entry for safe upgrade design. Incorporating these patterns proactively can help mitigate the long-term consequences of both known and anticipated SATD.

Implication 4.3) ***SATD in smart contracts can mislead front-end developers and introduce critical user-facing risks due to their limited familiarity with smart contract internals.*** In many DApp teams, front-end developers primarily focus on the user interface and may lack deep knowledge of Solidity, Ethereum's execution model, or the constraints of immutable smart contracts (Babushkin 2024). As a result, they may treat contract interfaces as stable APIs, assuming that exposed functions are complete, reliable, and safe to use. However, SATD, such as incomplete features, temporary workarounds, or unoptimized logic, violates these assumptions. Since SATD is typically documented only in source code comments, it remains invisible in ABIs or frontend libraries, creating a disconnect. This gap may lead front-end applications to unknowingly depend on fragile or unfinished logic, causing silent transaction failures, unexpected behavior, degraded performance, and reduced user trust. When SATD is left unresolved or becomes outdated (Observation 13), the risk increases significantly, as it can blur the line between past and present limitations, even for front-end developers who possess some familiarity with smart contract internals. To mitigate these issues, DApp teams must bridge the knowledge gap through clear documentation, ensure regular cleanup, cross-functional reviews, and educating front-end developers about the long-term

implications of SATD in immutable systems. Without such practices, SATD remains a hidden liability that can compromise both functionality and user experience in production DApps.

Implication 4.4) ***Public visibility of SATD through blockchain explorers can enhance transparency, inform reuse decisions, and support better risk assessment.*** Currently, self-admitted technical debt is discoverable only through manual inspection of smart contract comments. Yet, as demonstrated in our study, SATD is present in more than 10,000 real-world contracts, including those deployed in high-value or high-traffic settings (Section 4.4.3). Given the significant role that blockchain explorers like Etherscan play in the developer workflow, these platforms are well-positioned to improve transparency by making SATD visible to users and auditors. We recommend that explorers incorporate automated SATD detection features into their source code views. A simple starting point would be to flag contracts that contain common SATD indicators, such as the `TODO` keyword, especially in single-line comments, which account for over 85% of all identified instances (Observation 10). In addition to basic keyword matching, explorers could highlight the exact lines containing SATD, associate them with human-readable descriptions based on our taxonomy (Figure 4.6), and display warnings or risk annotations on affected contracts. This level of disclosure would allow contract consumers, including developers integrating third-party contracts and end users relying on DApp functionality, to make more informed decisions regarding contract trustworthiness and maintainability. For example, knowledge of a gas-inefficient SATD in a third-party contract might influence a DApp developer to isolate that component behind an upgradeable

interface. Ultimately, flagging SATD through blockchain explorers can contribute to a more transparent and accountable smart contract ecosystem.

4.8 Threats to Validity

In this section, we discuss potential threats to the validity of our findings, covering construct, internal, and external validity dimensions.

4.8.1 Construct Validity

Since Solidity is a relatively new programming language, we could not find any existing well-tested parser that supports our objective of comments extraction. As a result, we had to implement the parser ourselves. While we cannot guarantee that our parser is bug free, we tested it to the best of our abilities.

The vast majority of contracts in Ethereum are of little practical relevance (e.g., toy and test contracts) (Oliva et al. 2020). To mitigate this bias, we devise a heuristic based on activity level and currency influx to select real-world contracts. Although we carefully craft our heuristic and justify its parameter choices, we cannot guarantee that a better heuristic is nonexistent. We thus invite future research to replicate our study using other heuristics for filtering-in relevant, real-world contracts.

We used a pattern recognition tool called Jitterbug to detect SATD instances. Jitterbug is pre-trained on the code comments of 10 open source software projects (Maldonado et al. 2017a; Yu et al. 2020). While Yu et al. (2020) mentioned that their tool achieved a 100% precision in detecting SATD instances, this performance may

not hold in other domains. Our evaluation of false-positive and false-negative rates reported in Appendix A.1 indicates that Jitterbug performs well in the context of smart contracts. We also note that our findings are limited to the four SATD indication patterns that are detected by Jitterbug. Therefore, it is possible that we missed more complex (e.g., less obvious) forms of SATD.

We also used the Deckard tool for clone detection in smart contracts. A key parameter in Deckard is the *similarity threshold*. To mitigate construct validity threats, we employed a similarity threshold that has been deemed optimal by a careful sensitivity analysis conducted in a prior study (Kondo et al. 2020).

4.8.2 Internal Validity

To mitigate internal validity threats, we explicitly consider code cloning in Ethereum while studying SATD. In fact, given the strength of the connection between SATD and cloning, we devote an entire RQ to it (RQ2). Except for a few cases that we manually analyzed, we do not establish nor claim *causal* relationships between cloning and SATD. We also mitigate other internal validity threats by ensuring that our chosen statistical analysis procedures properly suit our goals and the characteristics of our datasets.

4.8.3 External Validity

The dataset used in this work is a collection of *all* verified smart contracts that were available on Etherscan as of March 18th, 2019. As such, our dataset is a complete representation of verified contracts at that point in time. Nevertheless, it is possible that smart contracts developed after such a period may have different

characteristics. It is thus possible that our results do not generalize to newer contracts. Furthermore, our results are unlikely to generalize to other programmable blockchain platforms.

To mitigate external validity threats in our manual classification of SATD types (RQ3), we followed open coding best practices and relied on the notion of theoretical saturation. From a statistics point of view, we highlight that the number of random SATD instances that we analyzed is larger than that of a statistically representative sample (95% confidence level, +-5 confidence interval). In addition, prior to classifying our SATD instances, we carefully reviewed the SATD taxonomies available in the literature (Rios et al. 2018; Bavota and Russo 2016; S. Maldonado and Shihab 2015; Maldonado et al. 2017a). Such a review informed and supported the entire coding process (especially the work of Rios et al. (2018)). Finally, we also leveraged our own expertise in smart contracts while performing the classification (Oliva et al. 2020; Kondo et al. 2020; Zarir et al. 2021; Oliva and Hassan 2021). We conclude from our manual study that real-world contracts have meaningful and likely impactful debts. Researchers should consider conducting online surveys and interviews to shed more light into the relevance and impact of SATD for different stakeholders (e.g., authors of the contract with SATD, contract developers that reuse a contract with SATD, and end-users).

4.9 Conclusion

Prior research has legitimately focused on detecting and studying unintentional security vulnerabilities in smart contracts. However, as general-purpose programs, smart contracts also encounter other maintainability challenges similar to those in

traditional software systems (Bosu et al. 2018; Chen et al. 2021c). One such challenge is technical debt Cunningham 1992, which is often intentional, introduced by developers as a conscious trade-off to meet short-term goals such as faster deployment, feature delivery, or adapting to rapidly evolving platforms.

This chapter set out to investigate the hypothesis that technical debt is significantly common in Ethereum smart contracts, focusing specifically on self-admitted technical debt (SATD). Our large-scale empirical study provides evidence supporting this hypothesis, showing that technical debt is not only present but also measurable and consequential within the ecosystem. We characterized SATD by determining its prevalence (RQ1), its association with code cloning (RQ2), and its types (RQ3). For RQ1, we found that 1.5% of contracts contain SATD, a prevalence that, while modest, suggests substantial maintainability and risk concerns. This finding is further corroborated by the occurrence of SATD in real-world, high-stakes contracts. In RQ2, we observed a strong connection between SATD prevalence and code cloning practices in Ethereum, which provides a key mechanism for its propagation. Finally, in RQ3, we derived a detailed taxonomy of SATD types (Figure 4.6) by manually investigating SATD instances through open coding and card sorting techniques, laying a foundation for future research.

Based on our findings, we discussed practical implications and outlined directions for future research (Section 4.7). Specifically, we: (i) identified solutions for SATD instances linked to recurring coding anti-patterns, such as hard-coded values and unbounded loops; (ii) emphasized the need to design DApps with built-in support for contract evolution mechanisms, addressing gaps between developer

needs and the Ethereum platform's progression; (iii) proposed integrating automated SATD detection into blockchain explorers like Etherscan to improve contract transparency and inform reuse decisions; and (iv) warned that unaddressed SATD can mislead frontend developers, introducing user-facing risks due to their limited familiarity with smart contract internals. We hope that our study will inspire and bootstrap additional maintainability studies on smart contracts, especially in the field of technical debt.

Beyond conducting the first empirical study on SATD, our work also addresses two key gaps highlighted by Chen et al. Chen et al. 2021c: (i) the lack of advanced software engineering techniques for detecting SATD in smart contracts by demonstrating the effectiveness of applying a state-of-the-art SATD detector (Yu et al. 2020) to smart contracts; and (ii) the lack of research data, by designing a scalable data collection approach capable of retrieving the source code of millions of smart contracts for the first time.

Finally, whether introduced intentionally or not, smart contracts require mechanisms to manage long-term maintainability issues, including SATD, bugs, and vulnerabilities. Given their immutable nature, developers often must incorporate architectural patterns early in DApp design to support post-deployment contract maintenance. Among the existing mechanisms (Section 2.14), the proxy pattern has been standardized in multiple well-audited and widely-used implementations (e.g., OpenZeppelin, Universal Upgradeable Proxy Standard), establishing it as a critical mechanism for upgradeable contract design. However, it also introduces significant risks such as reduced transparency, increased centralization, and new vulnerabilities or attack surfaces. As a result, studying proxy contracts is not only important

for understanding the long-term evolution of smart contracts but also for assessing their role as a potential risk vector in decentralized systems. In the following chapter, we present a novel and accurate method for detecting proxy-based smart contracts and conduct a large-scale empirical study on Ethereum. We then characterize proxy-based contracts in terms of their prevalence, integration strategies, and implementation types.

CHAPTER 5

A Large-Scale Exploratory Study on the Proxy Pattern in Ethereum

5.1 Introduction

A proxy is an entity that acts as an intermediary between two objects (Section 2.15). It serves as a representative for a target object, whether it is a network connection, a large memory object, a file, or other resource-intensive entities, without altering its interface. Hence, for a client, interacting with a proxy feels akin to using the actual target object. Proxies play a crucial role in software design, evidenced by their inclusion as one of the twenty-three standard design pattern (Gamma et al. 1994). For instance, proxies enhance modularity and encapsulation while offering functionalities like caching for resource-intensive operations, precondition checks,

and access control to the real object's operations (Shapiro 1986). Over time, proxy patterns have been widely adopted across various domains, including the internet, microservices (Richardson 2018), IoT systems (Bloom et al. 2018; Ngaogate 2019), and specifically blockchain applications (Wöhrer and Zdun 2018).

Ethereum (Sections 2.2), a leading blockchain platform, enables the deployment and execution of smart contracts (Sections 2.3), a type of software program that runs in decentralized trustless manner. Once these smart contracts are deployed on Ethereum, they become immutable (Section 2.5), meaning they cannot be patched post-deployment in responding to mitigating unintentional (e.g., bugs, vulnerabilities, etc.) and or intentional (e.g., technical debt, backdoor, etc.) internal quality issues. That said; however, just like any other traditional centralized software, smart contracts do require maintenance to stay secure, and functional over time (Lehman et al. 1997; Chen et al. 2021c; Chen et al. 2021a; Bosu et al. 2018). As a result, DApp (Section 2.4) developers face a crucial gap of how they reconcile the need for performing maintenance activities with the immutability of smart contracts post-deployment.

Since Ethereum lacks a native built-in mechanism for contract evolution (a.k.a., contract upgrades), to deal with the immutability of smart contracts and enable post-deployment contract evolution, thus facilitating maintenance activities, developers must proactively integrate architectural patterns into their smart contract designs prior to deployment. We briefly refer to such architectural patterns as contract evolution mechanisms (Section 2.14). Due to the importance of post-deployment maintenance, practitioners have devised various evolution mechanisms ranging from simple yet risky contract migration to the use of low-level EVM instructions

such as `SELFDESTRUCT` or `CREATE2`, as well as data separation-based approaches. Among the existing evolution mechanisms, the proxy pattern (Section 2.15), a type of data separation, has emerged as one of the most known mechanisms not only for enabling contract evolution but also enhancing modular design, and reducing operational costs. The popularity of the proxy pattern is particularly evident through sustained community efforts to formalize and refine proxy-related standards and to explore new applications over time (Jorge Izquierdo 2018; Peter Murray 2018a; Mudge 2018a; Gabriel Barros 2019b; Santiago Palladino 2019b; Mudge 2020a; pinkiebell 2021; Porter 2021; 0xAA et al. 2023; Shogo Ochiai 2023; Żuk 2024; Atarpara et al. 2024). Given this theoretical growth, we hypothesize that proxy pattern is being increasingly adopted in practice with the Ethereum ecosystem.

Within the blockchain context, the smart contracts that realize the proxy pattern are typically refereed to proxy contracts. In this setup, there exits two core structural elements; i) the proxy contract, ii) the logic (or implementation) contract. The former holds the persistent state (including a state variable that keeps logic contract address), whereas the latter contains the executable code. At runtime, when the client sends a transaction (Section 2.8) to the proxy, the proxy uses the low-level EVM `delegatecall` instruction (Section 2.11) to seamlessly forward the call to the logic contract, executing its code in the context of the proxy's storage. proxy contracts can be broadly categorized into two types: i) An Upgradeable Proxy Contract (UPC) is a proxy that explicitly supports (i.e., typically through function which is controlled by an authorized admin) changing its logic contract address over time, enabling logic contracts evolution; ii) a Forwarder Proxy Contract (FPC) is a proxy

that permanently forwards calls to a static fixed logic contract address, serving purposes like gas optimization, batching, or modular call routing (Section 2.15.5).

Proxy smart contracts enable flexible upgrades without redeployment, but they also introduce critical transparency and security risks. At the same time, they present unique research opportunities for understanding contract evolution and the socio-technical dynamics of decentralized systems.

- **Transparency Risks**. A key principle of blockchain systems is transparency: users, auditors, and automated tools rely on immutable, verifiable code to understand and trust contract behavior. Proxy contracts disrupt this principle by introducing an indirection layer that obfuscate the actual logic being executed (Pathade and Hooli 2025). This redirection makes it difficult to verify the system's behavior, trace its source, or even confirm the code path without deep technical expertise. For example, FPCs, while seemingly simpler in design, can also be exploited to deceive users by forwarding calls through unexpected or non-transparent paths, enabling phishing-like behaviors.

- **Violate Decentralization**. In the case of UPCs, these risks are magnified by the capacity for post-deployment logic contract changes. Such upgrades, though often justified for patching bugs, addressing technical debt, or adding features, can fundamentally alter the contract's behavior and invalidate previously held security assumptions. This creates significant security risks, as malicious upgrades or compromised governance can lead to unauthorized access, theft of assets, or systemic manipulation. Even with upgrade governance mechanisms (Section 2.9) in place, the inherent centralization in such governance contradicts the decentralized ethos of blockchain (Salehi et al. 2022).

- **Complicated Auditability.** The dynamic interplay between proxies and their logic contracts complicates code audits, static analysis, and formal verification. Not only is it difficult to identify whether a contract acts as a proxy, but it is even more challenging to determine the specific logic contract it delegates to, especially when delegation is dynamic or upgradeable. As a result, auditors must reason about multiple components and delegation paths, including potential edge cases, which increases the risk of oversight (Antonino et al. 2022).

- **Research Opportunities.** For researchers, proxy smart contracts represent a critical frontier in understanding how Ethereum applications evolve over time. As smart contract systems become more complex and long-lived, researchers are uniquely positioned to study the implications of contract evolution, from analyzing release engineering practices and detecting new vulnerabilities (Section 5.8.1.2) to building formal models that verify the correctness of multi-version systems. Furthermore, proxy contracts offer a rich context for examining the socio-technical aspects of blockchain governance, such as how upgrade rights are allocated, how decisions are made and communicated, and how user trust is maintained or eroded upon every upgrade.

Given these risks and research opportunities, it is imperative to develop scalable and accurate methods for detecting and studying proxy contracts. That said, significant gaps remain in both the tooling for detecting proxy contracts and our understanding of their prevalence, usage patterns, and structural variations. In particular, there is a lack of scalable and accurate techniques for identifying proxy contracts across the Ethereum blockchain. Existing approaches suffer from one or more key

limitations: they may be computationally expensive, rely heavily on source code artifacts (which are frequently unavailable), error prone, resource intensive or depend on heuristics that are prone to false positives and limited generalizability (Section 3.3.1).

To address these challenges, we propose a high-performing detection method that identifies active proxy contracts based on their intrinsic characteristics (Section 2.15.4). To validate our hypothesis, we apply this method in this chapter and conduct the first large-scale exploratory study aimed at systematically dissecting proxy contracts in Ethereum. Our analysis spans the entire history of Ethereum up to September 1, 2022, covering 50,845,833 deployed smart contracts and 1,695,517,186 transactions. Specifically, we investigate the following research questions:

Preliminary study: Is the proxy pattern prevalent enough in Ethereum to warrant large-scale study? Before committing to a full-scale analysis, it is important to establish whether proxy contracts are commonly used in Ethereum. A small-scope study helps assess the practical significance and motivates further investigation.

RQ1: How prevalent are proxy contracts in Ethereum? Understanding how frequently proxy contracts are used is essential to evaluating their impact on the Ethereum ecosystem. Quantifying their prevalence lays the foundation for deeper insights into smart contract development practices.

RQ2: What are the different creational patterns for deploying proxy contracts? Proxy contracts can be deployed through a variety of methods, each with trade-offs in transparency, security, and maintainability. Analyzing these patterns helps uncover common design practices and informs best-practice recommendations.

RQ3: What are the different types and properties of proxy contracts? Proxy contracts vary in structure and purpose, from enabling upgradeability to supporting modularization. Investigating these types and their alignment with standard implementations provides insight into developer choices and the evolution of proxy usage.

Contributions. To the best of our knowledge, this is the first in-depth, large-scale exploratory study of the proxy pattern in smart contracts. The value of our work lies not only in its findings and implications but also in its adaptable methodology and the novel research questions it raises for future exploration. Specifically, our key contributions are as follows: **(i)** we propose an efficient and accurate method for detecting active proxy contracts; **(ii)** we illustrate the prevalence of proxy contracts and their roles in the blockchain ecosystem from multiple perspectives; **(iii)** we mine various creational patterns for deploying proxy contracts and shed light on current practices; **(iv)** we analyze the prevalence of different types of proxy contracts based on their purposes and implementation strategies; **(v)** we publish four ground truth datasets to support the evaluation of proxy detection and proxy type classification methods; **(vi)** we examine the practical use of the proxy pattern in real-world decentralized applications (DApps); and **(vii)** we provide a detailed discussion on the advantages and disadvantages of the proxy pattern, different deployment styles, and the resulting research challenges and practical implications.

Chapter structure. Section 5.2 outlines our data collection process. Section 5.3 presents a preliminary analysis that motivates a deeper investigation of proxy patterns on Ethereum. Section 5.4 introduces our proxy detection method and its evaluation. Sections 5.5, 5.6, and 5.7 detail the motivation, methodology, and findings

for our three research questions. Section 6.8 discusses the benefits and drawbacks of adopting proxy patterns, along with a comparison of the two dominant deployment styles for proxy contracts. We then outline the broader implications of our findings and highlight opportunities for future research. Section 5.9 examines the threats to the validity of our study. Finally, Section 6.10 concludes this chapter.

5.2 Data Collection

This study requires comprehensive data on Ethereum smart contracts and their associated transactional activities. To gather this information, we utilized Google BigQuery, a fully managed, cloud-native data warehouse service provided by the Google Cloud Platform[1]. BigQuery is a commercial service with a limited free tier, which we leveraged for our data collection. Since forwarder proxies do not support upgrades, they are more predictable and secure by design, as the logic cannot be replaced by malicious actors after deployment. However, they offer no flexibility for post-deployment changes or enhancements.

Specifically, we extracted data from the `crypto_ethereum` dataset, which is part of the BigQuery Public Datasets program. This publicly available dataset includes detailed information on Ethereum blocks, transactions, smart contracts, events, tokens, and token holders. Data retrieval was performed using SQL-like queries, which can be executed through the BigQuery Console or programmatically accessed via the BigQuery API.

[1] https://console.cloud.google.com/

For our empirical analyses, we focused on two primary tables within the dataset: the `contracts` table[2] and the `traces` table[3]. The `contracts` table provides metadata about deployed smart contracts, including contract addresses, bytecode, and creation timestamps. In contrast, the `traces` table captures granular information about smart contract execution, including transaction hashes, input/output data, and execution results. We employed a snapshot of these tables covering data from August 7, 2015 (i.e., the official launch date of Ethereum) up to September 1, 2022 (i.e., the point at which we initiated our data analysis). As of that date, the `contracts` table contained records for 50,845,833 distinct smart contracts, while the `traces` table included data on 1,695,517,186 transactions and 5,503,071,306 execution traces.

5.3 Preliminary Study: Is the Proxy Pattern Prevalent Enough in Ethereum to Warrant Large-Scale Study?

5.3.1 Motivation

Before committing to a large-scale empirical study, we first perform a lightweight preliminary study to investigate whether there are observable traces of proxy usage in deployed smart contracts. This initial exploration helps us determine if the proxy pattern is a sufficiently common and practically significant practice to warrant deeper investigation.

[2] `bigquery-public-data.crypto_ethereum.contracts`
[3] `bigquery-public-data.crypto_ethereum.traces`

5.3.2 Approach

Given the role of proxies in enhancing modularity and encapsulation in software systems (Shapiro 1986), we first sought to explore whether such modular design principles are reflected in smart contract deployments. As a proxy pattern typically separates storage and logic across contracts, its usage implies a distribution of roles and responsibilities, an indicator of modularity. To investigate this, we examined Ethereum contract transactions for signs of cross-contract interactions, querying the `traces` table. Specifically, we calculated the monthly ratio of transactions that included at least one cross-contract call between distinct smart contracts, which we refer to as multi-contract transactions. These transactions may involve any number of internal calls, leveraging Solidity's cross-contract call types (Section 2.11).

Subsequently, since proxy contracts rely on the `delegatecall` instruction to forward execution while preserving the caller's context (OpenZeppelin 2018a), we analyzed the monthly distribution of `delegatecall` relative to other call types. To further validate the presence of actual proxy contracts, we utilized Etherscan's Proxy Verification tool[4]. We then automatically analyzed a statistically representative sample of multi-contract transactions containing at least one `delegatecall`. If multiple `delegatecall` operations were present in a transaction, one was selected at random.

We applied Cochran's sample size formula to accurately determine the needed sample size for representative results (Nanjundeswaraswamy and Divakar 2021). The inputs to this formula are confidence level and confidence interval (a.k.a., margin of error). In empirical software engineering research, a sample size is typically

[4] https://etherscan.io/proxyContractChecker

deemed representative (enough) when its associated confidence level and interval are 95% and 5 respectively (Pacheco et al. 2022; Lin et al. 2023; Tagra et al. 2022).

5.3.3 Findings

Observation 5.1) ***Multi-contract transactions have become increasingly prevalent, with nearly one-third of transactions involving interactions between multiple contracts by 2022.*** Figure 5.1 illustrates a notable rise in the ratio of multi-contract transactions. Initially, in 2015, multi-contract transactions were virtually non-existent, but a dramatic surge was seen by the end of the year, with December hitting a notable 16.54% ratio. Throughout 2016 to 2019, the ratio meandered between 4% to 17%, reflecting some fluctuations but a general acceptance and steady usage of multi-contract transactions. However, 2020 marked a significant year with an average ratio soaring past 25%, and by August reaching an astonishing 32.91%. This uptrend persisted into 2021, maintaining an average above 30%. As of the recent data in 2022, the trend continues to hover around 30%, suggesting a consistent adoption rate and reliance on multi-contract transactions in the Ethereum ecosystem. This long-term observation underscores the growing complexity and interconnectedness of contracts on the platform over the years.

Observation 5.2) ***The prevalence of multi-contract transactions involving the*** `delegatecall` ***operation, a key indicator of proxy contract usage, has shown a significant upward trajectory.*** As decentralized applications grow in complexity and modularity, developers increasingly avoid hardcoding all inter-contract communications using regular `call` operations. Solidity's execution model complicates

matters, as a `call` modifies the callee's state, which may not align with application requirements. Some scenarios demand updates to the caller's state or no state changes at all, necessitating alternative cross-contract call types (Section 2.11).

Figure 5.2 presents the monthly ratio of multi-contract transactions by call type from August 2015 to September 2022. Initially, `call` dominated, accounting for over 85% of multi-contract calls in 2015 and remaining dominant through 2017. However, its usage gradually declined, stabilizing around the mid-70% range by 2022. Meanwhile, `delegatecall` remained negligible until May 2016 but exhibited continuous growth thereafter, exceeding 70% of multi-contract transactions by late 2022. The use of `staticcall`, introduced in October 2017, remained low until mid-2018, after which it rose steadily and became a common call type. In contrast, `callcode` showed early activity in 2015 but quickly faded in popularity, becoming nearly obsolete after 2017.

Observation 5.3) ***98.4% of multi-contract transactions with delegatecall operations involve a proxy contract.*** The increasing use of `delegatecall` operations prompted our investigation. From a dataset of 190,493,052 multi-contract transactions containing at least one `delegatecall`, we drew a statistically representative random sample. Using a confidence level and interval of 95% of 5 respectively (sampling parameters), we obtained 385 `delegatecall` operations. For each `delegatecall`, we identified the contract that initiated the trace, referred to as the caller contract. We categorized the caller contract as either proxy or non-proxy using Etherscan's Proxy Verification service. We found that proxy contracts are predominant (98.4%), with the remaining 1.6% involving no proxy contracts. This high prevalence of proxies motivates us to conduct a more comprehensive

study, exploring their prevalence, creation patterns, and types in further detail (Sections 5.5, 5.6 and 5.7).

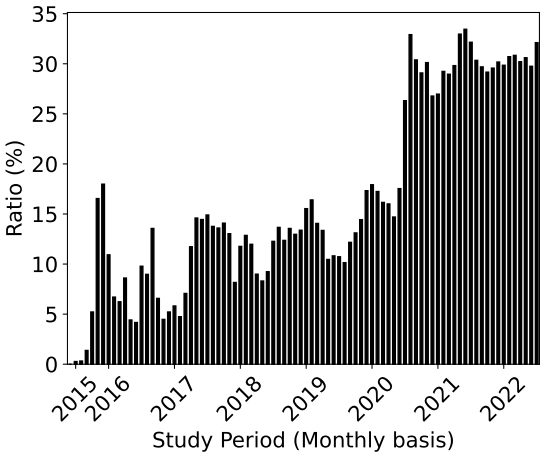


Figure 5.1: Monthly ratio of multi-contract transactions.

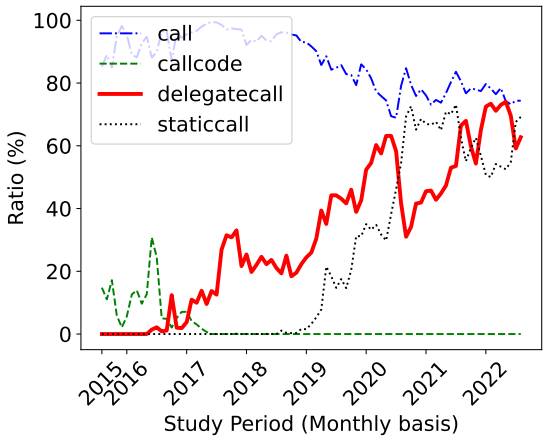


Figure 5.2: Monthly ratio of multi-contract transactions by call type.

Summary

Preliminary Study: Is the Proxy Pattern Prevalent Enough in Ethereum to Warrant Large-Scale Study?

Multi-contract transactions have steadily increased, reaching 33% of all transactions by 2022. Among these, the use of `delegatecall` has surged, 98.4% of such transactions involve a proxy contract, highlighting their growing popularity in Ethereum ecosystem.

5.4 The Proposed Proxy Detector

In this section, we introduce our proxy detection approach (Section 5.4.1) and subsequently evaluate its performance (Section 5.4.2).

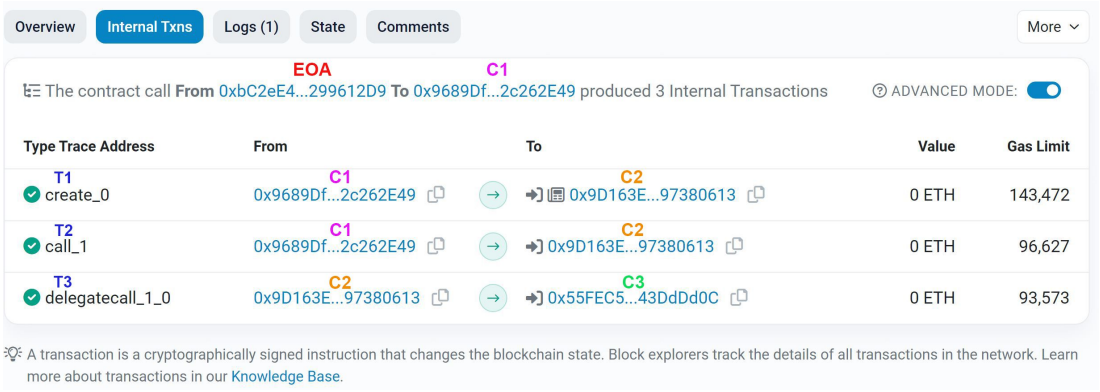


Figure 5.3: Trace table of example transaction *T*.

5.4.1 Approach

In our preliminary study (Section 5.3), we initially used Etherscan's Proxy Verification service to identify proxies. This service works by heuristically detecting known proxy patterns in a contract's bytecode, typically through the presence of low-level `delegatecall` instructions. However, this approach has notable downsides. For instance, it may misclassify contracts (e.g., libraries using `delegatecall`), fails for custom proxies, and does not guarantee that the displayed implementation is what the proxy currently uses, leaving room for deceptive or malicious behavior. Thus, as we scaled up our analyses for the remainder of this chapter, we found this service to be impractical, costly, inaccurate. Therefore, this prompted us to develop a more efficient and accurate approach.

Consider the trace table for the example transaction *T* shown in Figure 5.3 (Section 2.8 for a detailed explanation of this example). To detect proxy contracts, we rely on the inherent properties of a proxy contract (Section 2.15.4) and match historical transactions of deployed contracts against these properties.

More specifically, we first filter in all traces that involve a `delegatecall` operation between two contracts (i.e., Property #1). For each identified `delegatecall` trace, we label its caller and callee contracts as the "candidate proxy" and "candidate logic" contracts, respectively. For instance, in trace *T3* from Figure 2.1, *C2* and *C3* serve as the candidate proxy and candidate logic contracts.

To confirm whether a candidate proxy is indeed a proxy contract, we check if it preserves the interface of the candidate logic contract. This is done by examining the preceding trace of the `delegatecall` trace. The preceding trace is the immediate parent trace of the `delegatecall` trace in the trace call graph, here *T2*. We then compare the calldata of *T2* and *T3*. If the first four bytes of the calldata in both traces match, it indicates that the proxy is forwarding the same function call to the logic contract without altering the input. These first four bytes represent the function selector, derived from the first four bytes of the Keccak-256 hash of the function signature (e.g., `transfer(address,uint256)`). This selector uniquely identifies the invoked function. A match between the selectors in the parent trace (*T2*) and the `delegatecall` trace (*T3*) confirms that the proxy contract exposes and preserves the interface of the logic contract, thus satisfying Property #2.

We operationalized the above procedure into a BigQuery SQL query and applied it across historical transaction traces of all 50,845,833 deployed smart contracts using the Google BigQuery console, a web-based platform that enables scalable querying over Ethereum's public dataset hosted on Google Cloud. This approach allowed us to efficiently analyze millions of transaction traces and systematically detect proxy contracts at scale, without relying on heuristic-based methods, transaction replaying, or manual inspection (Section 3.3.1). In addition, by leveraging

two core conceptual properties inherent to proxy contracts (Section 2.15.4) our approach generalizes beyond conventional fallback-based proxies, capturing a broader and more diverse set of proxy contracts.

The output of this process is a comprehensive list of all <proxy contract, logic contract> pairs. Note that a proxy contract may be associated with multiple logic contracts over time (e.g., due to contract evolution or modularization). Our approach is capable of identifying all logic contracts that a proxy contract has ever interacted with.

5.4.2 Evaluation

To evaluate our proxy detection approach, we followed these key steps. Initially, we created a ground truth dataset by selecting a statistically representative sample from a pool of all deployed contract. The sample size, determined with a 95% confidence level and a confidence interval of 5, amounted to n = 385.

Second, we conducted an analysis of the decompiled *bytecode* of the sampled contracts to categorize them as either "proxy" or "non-proxy". To achieve this, we manually inspected the decompiled bytecode rather than relying on transactional activities as some contracts may be inactive. Specifically, we obtained the bytecode for all sampled contracts from the Ethereum public dataset's `contracts` table. Subsequently, we used the Panoramix[5] tool for bytecode decompilation.

In identifying proxy contracts, we again analyze the each decompiled bytecode against the inherent properties of a proxy contract (Section 2.15.4). First, focused

[5] https://github.com/palkeo/panoramix

```
1  def storage:
2    stor0 is addr at storage 0
3
4  def _fallback() payable: # default function
5    delegate stor0 with: # 1) delegatecall
6      funct call.data[0 len 4] # 2) similar interface
7      gas gas_remaining wei
8      args call.data[4 len calldata.size - 4]
9    if not delegate.return_code:
10      revert with ext_call.return_data[0 len return_data.size]
11   return ext_call.return_data[0 len return_data.size]
```

Listing 5.1: An example of a proxy contract's decompiled bytecode.

on the presence of the `delegatecall` statement (i.e., Property #1). Contracts lacking this statement in their decompilied bytecode were categorized as non-proxy by definition (OpenZeppelin 2018a), resulting in 290 such contracts. For the remaining 95 sample contracts, we analyzed the `delegatecall` statement parameters to determine if the function being called via delegation exhibited a similar function selector to the function where the `delegatecall` statement was originally found in the sample contract (i.e., Property #2). We accomplished this by examining the use of the first four bytes (i.e., the function selector) of the `calldata`. Contracts where this condition was met were labeled as proxy (90 instances) while those that did not meet this criterion were labeled as non-proxy (5 instances). Listing 5.1 shows an example of a proxy contract's bytecode[6]. Line 5 shows the presence of the delegatecall statement, while line 6 shows that the call uses the first four bytes of the input `calldata`.

[6]Etherscan bytecode decompiler for contract 0x7f27...d7f5

Table 5.1: The performance of our proxy detection approach.

	Precision	**Recall**	**F1-measure**
Proxy	100%	68.9%	81.6%
Other	91.3%	100%	95.5%

In total, our ground truth contains 90 (23.4%) proxy contracts and 295 (76.6%) non-proxy contracts. Further details about our ground truth dataset can be found in our supplementary material package.

We then applied our proxy detection method to our ground truth dataset and evaluated its accuracy using precision, recall, and the F1-measure (Table 5.1). Our approach achieved perfect precision (100%), but the recall was 68.1%, indicating that 28 proxy contracts were missed. The resulting F1-measure was 81.6% for proxy detection and 95.5% for non-proxy cases.

A closer examination of the 28 (7.27% of the sampled contracts) missed cases revealed a common pattern: in 26 of them, both the decompiled bytecode and the available source code indicated the presence of a proxy contract, but the proxy had not been involved in any transactions, it remained *inactive* since deployment. In the remaining two cases, although each received one or more transactions, those transactions did not trigger the proxy's functionality. Since our detection method relies on analyzing a contract's transactional activity, such inactive cases remained undetected. It is worth noting, however, our approach demonstrated 100% precision and recall in detecting active proxy contracts (i.e., proxy contracts whose proxy functionality is invoked at least once after deployment) which is the primary focus of our study.

Finally, we conducted a comparative analysis of our detection approach with a related study by Salehi et al. (2022), who identified 1,427,215 proxies between September 5, 2020, and July 20, 2021. In contrast, our approach detected 1,723,309 proxy contracts within the same time frame, 296,094 more than those reported by Salehi et al. (2022). Unfortunately, due to the unavailability of their proxy dataset, we were unable to perform a direct one-to-one comparison. While both approaches rely on the behavioral characteristics (i.e., analyzing the runtime behavior) contracts for proxy detection, meaning neither can identify inactive proxies, there are notable differences.

First, Salehi et al. (2022) focus primarily on `fallback`-based proxies and rely on transaction replaying, which involves re-executing historical transactions to observe contract behavior. This technique requires reconstructing the precise blockchain state at the time of execution, making it computationally expensive, difficult to scale, and potentially error-prone. Transaction replaying can fail to reproduce original behaviors due to missing state data, reliance on non-deterministic variables (e.g., timestamps, gas costs), or changes in external contract dependencies. In contrast, our approach avoids transaction replaying entirely by leveraging and mining publicly available historical transaction traces. Second, by analyzing two inherent conceptual proprieties of proxy contracts, our method generalizes beyond `fallback`-based proxies and enables more scalable, robust, and accurate detection. Despite the inherent limitations in directly comparing the results, our evaluation on a ground truth dataset of 385 contracts and the high precision achieved provide strong confidence in the effectiveness of our approach.

Summary

The Proposed Proxy Detector

We introduced an approach to identify proxy contracts based on their runtime behavior. Our evaluation, using a ground truth dataset of active proxy contracts, shows that our method achieves perfect precision with a 68.9% recall rate. Our focus in this study was on active proxy contracts, those with at least one instance of proxy functionality usage, resulting in both perfect precision and recall for our proposed approach.

5.5 RQ1: How Prevalent Are Proxy Contracts in Ethereum?

5.5.1 Motivation

The preliminary study (Section 5.3) revealed a significant and growing presence of proxy contracts in Ethereum. Specifically, we observed that by 2022, nearly one-third of all Ethereum transactions involved multiple contracts, with the `delegatecall` operation, central to proxy behavior, accounting for over 70% of multi-contract interactions. Furthermore, 98.4% of sampled transactions involving `delegatecall` were confirmed to originate from proxy contracts, as verified using Etherscan's Proxy Verification tool.

These findings provide compelling evidence that the proxy pattern is not only widespread but also deeply integrated into the execution model of modern smart contracts. Motivated by this strong preliminary signal, we now turn to a comprehensive, large-scale investigation to measure the overall prevalence of proxy contracts across the Ethereum blockchain, spanning from its inception to the present.

5.5.2 Approach

To detect proxy contracts, we analyze the runtime behavior of over 50 million smart contracts collected in Section 5.2, using the method described in Section 5.4.1. After identifying the overall prevalence of proxy contract instances, we perform four separate analyses to examine their prevalence from different perspectives.

Activity Level. The goal of this analysis is to assess the activity levels of proxy contracts. We also compare these levels with those of non-proxy contracts (i.e., any contract that is not a proxy). Contract activity is measured based on the number of inbound transactions. To compare the distributions of inbound transactions for proxy and non-proxy contracts as of September 2022, we use the Complementary Cumulative Distribution Function (CCDF). In a CCDF graph, the y-axis represents the percentage of data points at or above a specific value, while the x-axis shows the variable of interest, in this case, the number of inbound transactions. To statistically assess whether proxy contracts receive more inbound transactions than non-proxy contracts, we apply the non-parametric one-tailed Mann–Whitney U test ($\alpha = 0.05$). Additionally, we evaluate the magnitude of the difference using Cliff's Delta (δ), interpreting the effect size according to the thresholds defined by Romano et al. (2006): *negligible* if $|\delta| \leq 0.147$, *small* for $0.147 < |\delta| \leq 0.33$, *medium* for $0.33 < |\delta| \leq 0.474$, and *large* for $|\delta| > 0.474$.

Stakeholder Adoption. The goal of this analysis is to determine the engagement of EOAs (Externally Owned Accounts, a.k.a. stakeholders) in the creation of proxy contracts. We analyzed both the count and the ratio of EOAs that initiated proxy creation transactions on a monthly basis. An EOA may either deploy a proxy contract directly or request its deployment via a third-party contract (e.g., a proxy

factory). In both cases, the EOA initiates the contract creation transaction, allowing us to identify its unique address. More specifically, for each given i^{th} month, we counted the total number of EOAs that had initiated at least one proxy contract creation transaction from August 7, 2015, up to the end of the ith month. We also recorded the total number of EOAs that had initiated any contract creation transaction (proxy or otherwise) by the same point in time. The ratio of EOAs involved in proxy creation was then computed by dividing the number of EOAs who had initiated proxy deployments by the total number of EOAs who had initiated any contract deployments as of that month.

Use Case. The goal of this analysis is to identify the distinct use cases in which proxy contracts are employed. We define a use case as a set of similar proxy contracts deployed by the same entity for a common purpose. Specifically, we analyze the monthly count of such use cases over time. To identify these use cases, we follow a seven-step process:

1. For each detected proxy contract, we extract its deployer address, also referred to as the caller address of the proxy creation trace (e.g., *C1* in Figure 5.3). This deployer address represents the entity (e.g., a developer or a smart contract) responsible for deploying the proxy.
2. We collect the proxy's associated logic contracts. This step is essential because developers, often within the same organization, may reuse or clone proxy contracts for different purposes. To capture this variability in intent, we include the addresses of all logic contracts to which the proxy has ever delegated calls, as these define the behavior behind each proxy.

3. Using the proxy address, proxy bytecode, logic contracts, and deployer address, we cluster proxy contracts based on their bytecode similarity and deployer address. This enables us to group together similar proxy contract deployed by the same entity.
4. Within each cluster, we construct a graph in which nodes represent either proxy contract addresses (proxy nodes) or their associated logic contract addresses (logic nodes). An edge is drawn between a proxy node and a logic node if the proxy delegates calls to that logic contract.
5. We identify connected components within each graph. Each component corresponds to a unique use case. A connected component represents a set of similar proxy contracts that have interacted with the same or related logic contracts, and therefore likely serve a common purpose.
6. Within each use case, we retain only the earliest deployed proxy contract, recording its creation timestamp as the use case's starting point, and discard any subsequent duplicates.
7. Finally, we compute the monthly count of distinct usage cases based on the timestamps of the retained proxies.

Proxy Transaction Flow. The goal of this analysis is to shed light on the usage of proxy contracts by examining smart contract transactions over time. While we previously showed that 98.4% of a sample of multi-contract transactions involving `delegatecall` included a proxy contract (Section 5.3), we did not investigate how proxy usage has evolved due to the scalability limitations of Etherscan's Proxy Verification tool. Our proposed proxy detection approach addresses this limitation, enabling us to perform a longitudinal analysis. To this end, we first examine the

monthly ratio of transactions that involve proxy contracts. Then, building on our earlier analysis of multi-contract transactions (Section 5.3), we compute a refined metric: the monthly ratio of multi-contract transactions in which at least one proxy contract participates. Since transactions reflect how contracts interact on-chain, this metric offers insight into the growing role of proxies in smart contract design and execution.

5.5.3 Findings

Observation 5.4) ***Over 14% of all deployed smart contracts are proxy contracts.*** As of September 2022, a total of 50,845,833 smart contracts had been deployed on Ethereum. Among these, 7,241,339 contracts (14.2%) are identified as active proxy contracts. Each proxy is paired with at least one logic contract, although a proxy may be associated with multiple logic contracts over time to accommodate contract upgrades or evolving functional needs. Some proxies also act as routers, dynamically directing requests to appropriate logic contracts, enhancing modularity. A prominent example is the ERC-2535 Diamond Proxy Pattern (Mudge 2020a), in which a proxy interacts with multiple facet contracts. In total, we identified 7,296,032 proxy–logic contract pairs. Notably, our detection pipeline, operating on over 1.6 billion historical transactions, completed this large-scale analysis in approximately 17 minutes, demonstrating both the scalability and efficiency of our approach.

Figure 5.4 shows the CCDF of the number of logic contracts a proxy has interacted with over its lifetime. The analysis reveals that the vast majority of proxy contracts (99.56%) are linked to only a single logic contract. Toward the tail of the

distribution, the probability of proxies having multiple logic contracts drops sharply, falling below 10^{-4}% at $N = 100$ and continuing to decline thereafter.

An extreme outlier is the proxy contract `BZxProxy`[7], maintained by the bZx Protocol[8], which has been associated with 141 distinct logic contracts. This proxy maintains a mapping of services to their corresponding logic contracts and delegates calls accordingly, effectively functioning as a router. It also reflects an advanced case of contract evolution, where different versions or implementations of services are incrementally added, updated, or replaced over time. This design allows for fine-grained contract evolution and modular service deployment within a single proxy architecture.

Observation 5.5) ***Proxy contracts exhibit a statistically higher activity level than non-proxy contracts.*** Figure 5.5 displays the CCDFs of the number of inbound transactions for proxy and non-proxy contracts. The black line represents proxy contracts, and the red dashed line represents non-proxy contracts. The distribution follows a long-tail pattern, indicating that while most proxy contracts receive relatively few transactions, a small subset receives a disproportionately high number.

In the transaction range $1 \leq N \leq 256$, proxy contracts consistently show a higher likelihood of receiving more inbound transactions compared to non-proxy contracts. Beyond this range ($N > 256$), non-proxy contracts begin to dominate slightly in terms of transaction count. However, this tail segment accounts for only a small fraction of the contract population in both categories.

At the extreme end of the distribution, the maximum number of inbound transactions is observed among non-proxy contracts. Nevertheless, a one-tailed Mann–Whitney

[7] https://etherscan.io/address/0x1cf226e9413addaf22412a2e182f9c0de44af002
[8] https://b0x.network/

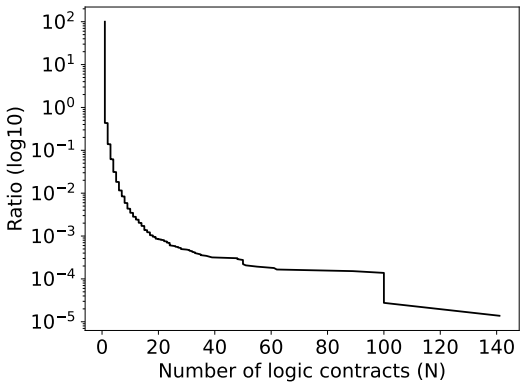


Figure 5.4: CCDFs of the number of logic contracts per proxy.

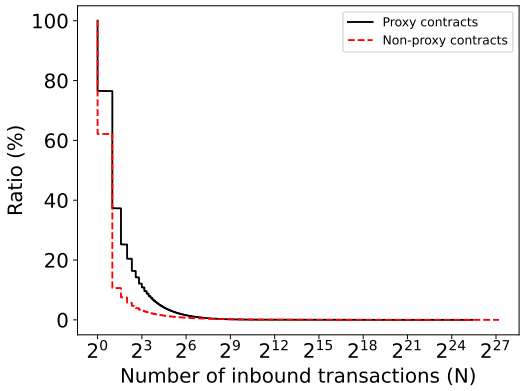


Figure 5.5: CCDFs of the number of inbound transactions.

U test confirms that the overall distributional difference is statistically significant (p-value < 0.05). Additionally, Cliff's Delta ($\delta = 0.34$) indicates a medium effect size.

Observation 5.6) ***There is an upward trend in the number of EOAs initiating proxy contract transactions, surpassing two-thirds of all EOAs who deployed any contract by 2022.*** As shown in Figure 5.6, the number of EOAs that instantiated at least one contract (red line) gradually increased over time, reaching 3,636,192 by the end of the study period. EOAs began adopting proxy contracts in mid-2016 (black line), coinciding with the instantiation of the first proxy contract (Observation 6). Since then, the number of EOAs deploying proxies has grown rapidly, peaking at 2,487,994 as of September 2022. The narrowing gray area between the red and black lines indicates that, although the gap between EOAs deploying any contract and those deploying proxy contracts was initially large, it significantly diminished over time. This trend reflects a growing tendency among EOAs to use proxy contracts.

Additionally, Figure 5.7 illustrates the monthly ratio of EOAs that deployed proxy contracts. This ratio remained relatively stable with modest growth until

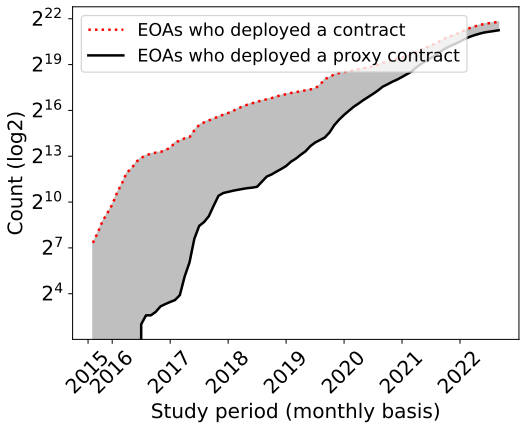


Figure 5.6: The count of EOAs deployed contracts as of each month.

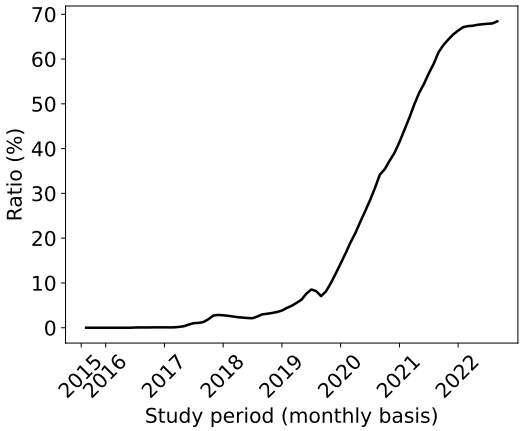


Figure 5.7: Ratio of EOAs who deployed a proxy as of each month.

mid-2019, after which it exhibited a consistent upward trend. By the end of the study period, it plateaued at 68.4%, meaning that as of September 2022, more than two-thirds of EOAs that deployed any contract had deployed at least one proxy contract.

Observation 5.7) ***The number of distinct proxy usage cases has steadily increased, reaching nearly 80,000 by September 2022.*** Figure 5.8 shows a clear upward trend in the creation of distinct use cases over time, with notable acceleration beginning in 2017. The early years (2015–2016) exhibit minimal activity, but subsequent years display consistent growth, peaking in August 2022 with over 2^{13} new use cases in a single month. Although monthly fluctuations are present, the overall trajectory indicates increasing adoption of proxy contracts across a wide range of use cases. Since each usage cases represents a different functional deployment of proxies, this growth reflects the expanding popularity and versatility of the proxy pattern in real-world smart contract development. The graph also reveals that the proxy pattern has been in use since the early days of Ethereum. The

first recorded usage cases appeared in mid-2016, approximately 10 months after Ethereum's launch. In total, we identified 79,171 distinct proxy use cases during the study period, underscoring the growing practical significance of proxies across diverse development scenarios.

Observation 5.8) ***Over 20% of transactions in 2022 involved a proxy contract.*** Figure 5.9 presents the monthly ratio of transactions involving a proxy contract. From a transactional perspective, this trend reflects a significant portion of smart contract activity on the blockchain is being routed through proxy contracts. Notably, proxy contracts were involved in more than 20% of all transactions in 2022, underscoring their increasing operational relevance on-chain. The importance of proxies becomes even more apparent in the context of fine-grained and modularized contract architectures. As shown in Figure 5.10, the monthly ratio of multi-contract transactions that include at least one proxy contract has continued to rise. Despite some fluctuations, proxy involvement in these complex transaction patterns reached nearly 70% by 2022. This highlights the essential role of proxy contracts in supporting modular smart contract designs, particularly in large-scale decentralized applications.

Summary

RQ1: How Prevalent Are Proxy Contracts in Ethereum?

Over 14% of deployed ocntracts are active proxy contracts. Our assessment from the four viewpoints of activity level, use case, stakeholder adoption, and proxy transaction flow shows that the tendency for using proxy contracts is growing. We also confirm the substantial relevance of proxy contracts in modular application designs.

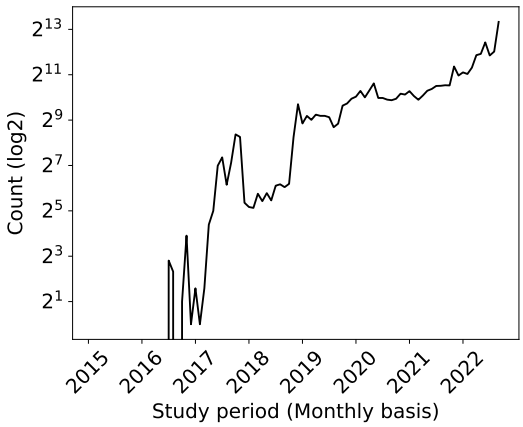


Figure 5.8: Monthly number of distinct use cases.

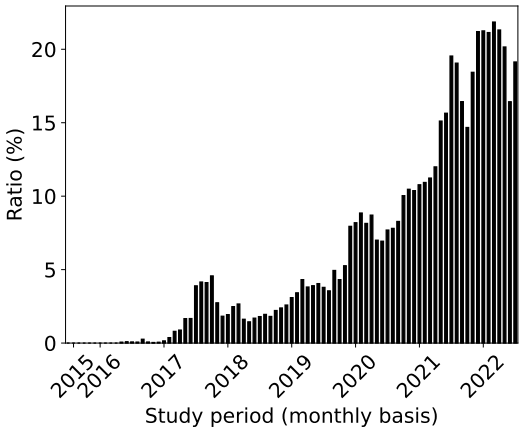


Figure 5.9: Monthly ratio of transactions that involve at least one proxy.

5.6 RQ2: What Are the Different Creational Patterns for Deploying Proxy Contracts?

5.6.1 Motivation

Proxy contracts play a central role in enabling modularity and upgradeability in Ethereum-based decentralized applications (DApps). As previously observed (Observation 4), proxies account for over 7.2 million contracts, approximately 14% of all deployments, highlighting their status not as niche mechanisms, but as foundational elements of modern smart contract infrastructure.

Despite their widespread adoption, the manner in which proxy contracts are deployed (i.e., the creational patterns) remains underexplored. These patterns reflect critical architectural decisions, particularly whether a proxy is deployed directly by

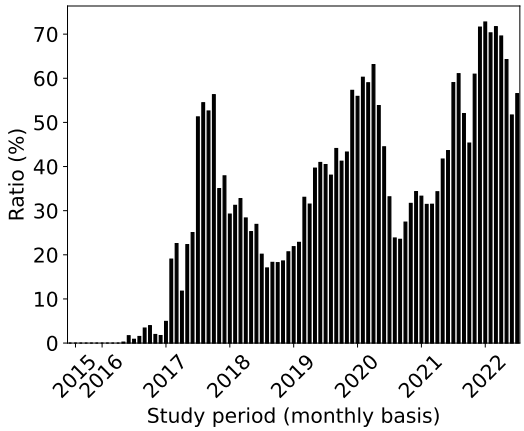


Figure 5.10: The monthly ratio of multi-contract transactions that involve at least one proxy contract.

an externally owned account (EOA) or programmatically through other smart contracts (CAs). Each approach carries different implications for transparency, maintainability, security, and cost-efficiency. Hence, understanding the diversity and prevalence of these deployment strategies offers valuable insights into the design practices and engineering trade-offs within the Ethereum ecosystem. Much like software design patterns guide modular development in traditional systems, creational patterns in smart contracts can inform developers, tool builders, and auditors in selecting or recommending deployment strategies that promote robust, scalable, and auditable DApp architectures.

5.6.2 Approach

To identify the creational patterns used in proxy deployment, we conducted a mining process to trace the underlying deployment infrastructure. Specifically, for each

proxy, we retrieved its contract creation transaction and examined the corresponding trace table (Section 2.8) to locate the trace associated with the proxy's creation. This trace had to satisfy two criteria: (1) its type had to be `create`, and (2) the callee address had to match the proxy contract's address.

Next, we extracted the caller address from this trace, which we referred to as the "deployer address". We then determined whether the deployer address corresponded to an Externally Owned Account (EOA) or a Contract Account (CA) (Section 2.7). This was achieved by cross-referencing the deployer address with all known contract addresses, obtained by querying the `contracts` table from the Ethereum public dataset (Section 5.2). If the deployer was a CA, we recorded it and recursively repeated the same process to trace its creator until we identified the root EOA, i.e., the original entity responsible for deploying the first contract in the proxy deployment infrastructure. Conversely, if the deployer address was an EOA, we recorded it as the entity that directly deployed the proxy contract. Once the root EOA was identified, we constructed a sequence of contract creation steps beginning with the root EOA and ending with the proxy contract.

To extract creational patterns, we then assigned labels to each node in the sequence. The initial node was labeled as the root EOA. Any intermediate contract that created another contract at runtime was labeled as a factory contract (FA). If an intermediate factory also served as a proxy contract, it was labeled as a proxy factory (PF). Proxy factories were labeled as such to capture cases where the deployment logic itself can be upgraded through the proxy pattern evolution mechanisms or reused via proxy delegation, allowing for shared, modular factory logic across different use cases. The final node in the sequence was labeled as a proxy (P). This

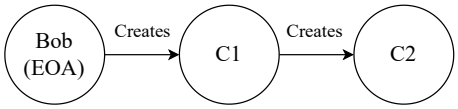


Figure 5.11: Example of sequence of contract creations.

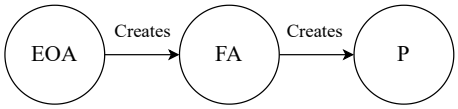


Figure 5.12: Example of labeled sequence of contract creations.

labeled sequence captured the structure of the deployment and was referred to as a "proxy creational pattern".

For instance, consider a scenario where Bob (the root EOA) creates a factory contract C1, which subsequently creates a proxy contract C2. Figures 5.11 and 5.12 illustrate the sequence of contract creations and the corresponding labeled pattern for this example.

After obtaining the creational patterns, we conducted a series of analyses. First, we identified the distinct proxy creational patterns and developed a metamodel using a UML class diagram to succinctly represent the various methods employed for proxy deployment. We then analyzed the prevalence of each creational pattern across different use cases (Section 5.5).

Second, we investigated how the nature of the deployer, whether proxies were deployed directly by an externally owned account (EOA) or programmatically by other smart contracts (CAs), affects the scale and structure of use cases. For each of the 79,171 identified use cases (Section 4.4), we categorized them based on the nature of their proxy deployer and computed their size in terms of the number of proxy contracts (N). We then compared the complementary cumulative distribution functions (CCDFs) of use case sizes across these categories. Additionally, we employed a one-tailed Mann–Whitney U test ($\alpha = 0.05$) to evaluate whether use cases

with proxies deployed programmatically tend to be larger than those with direct EOA deployment.

Finally, we classified use cases into singleton (N = 1) or non-singleton (N > 1) groups and conducted a chi-square test to assess the relationship between use case size and the nature of the deployer. We quantified the effect size using the Phi (ϕ) coefficient, interpreted as follows: *small* if $0.0 < |\phi| \leq 0.3$, *medium* for $0.3 < |\phi| \leq 0.5$, and *large* for values exceeding 0.5.

5.6.3 Findings

Observation 5.9) ***We found 12 different creational patterns for deploying proxy contracts.*** A total of 7.2 million proxies are deployed through these 12 creational patterns. The patterns, as shown in Table 5.2, are read from left to right, where the "$\rightarrow$" operator indicates a deployment relationship: the left-hand side represents the deployer type, and the right-hand side represents the deployee type within a pattern. Figure 5.13 presents a metamodel that summarizes the 12 detected creational patterns using a UML class diagram.

All patterns are ultimately initiated by an EOA, since only EOAs can initiate Ethereum transactions (Section 2.8). Specifically, an EOA can either directly deploy instances of *proxy contracts* (P) or first deploy a *factory contract* (FA). A factory contract is a smart contract designed to create other contracts at runtime, often following a predefined template. It typically encapsulates the deployment logic and allows developers to efficiently instantiate multiple similar contracts, making it a common pattern for modular and scalable deployments. Inspired by the composite design pattern, the metamodel illustrates how a factory (i.e., the composite) can

deploy either other factory contracts or proxy contracts (i.e., the leaves). Additionally, there exists a specialized type of factory contract, referred to as a *proxy factory* (PF), in which the factory contract itself is also implemented as a proxy contract. This design enabling reuses of logic across different use cases or future changes to the factory's behavior or configuration (e.g., modifying the template contract) as needed.

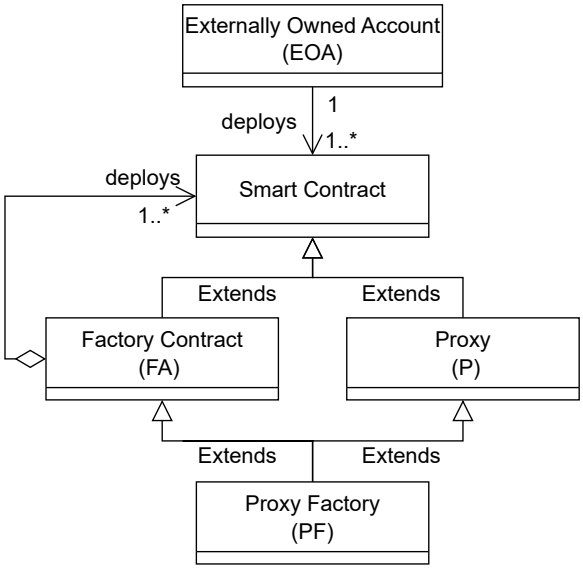


Figure 5.13: Metamodel of the 12 creational patterns for proxy contract deployment.

Observation 5.10) ***The 12 proxy creational patterns can be classified into two major deployment styles based on the nature of the deployer: i) on-chain and ii) off-chain.*** On-chain refers to a style in which proxy contracts are deployed by smart contracts (CAs) at runtime, encompassing 11 of the 12 patterns. In contrast, off-chain style is characterized by proxy contract being deployed by an externally owned account (EOA) through scripts that are maintained *outside* the blockchain.

Table 5.2: Overview of the 12 detected proxy contract creational patterns sorted based on the category and the proxy instance count.

Id	Creational Pattern	Category	Use Case Count	Proxy Instance Count
1	EOA→P	Off-chain	30,933 (39.07%)	50,174 (0.69%)
2	EOA→FA→P	On-chain	31,520 (39.81%)	6,618,012 (91.39%)
3	EOA→PF→P	On-chain	565 (0.71%)	379,293 (5.24%)
4	EOA→FA→FA→P	On-chain	182 (0.23%)	123,349 (1.70%)
5	EOA→FA→FA→FA→FA→P	On-chain	7 (0%)	35,764 (0.49%)
6	EOA→FA→PF→P	On-chain	15,627 (19.74%)	31,586 (0.44%)
7	EOA→PF→PF→P	On-chain	138 (0.17%)	1,968 (0.03%)
8	EOA→FA→PF→PF→P	On-chain	183 (0.23%)	589 (0.01%)
9	EOA→FA→FA→FA→P	On-chain	11 (0.01%)	599 (0.01%)
10	EOA→FA→FA→PF→P	On-chain	3 (0.01%)	3 (0.00%)
11	EOA→FA→FA→PF→PF→P	On-chain	1 (0%)	1 (0.00%)
12	EOA→FA→FA→PF→PF→PF→P	On-chain	1 (0%)	1 (0.00%)

Although proxy instances from both styles are ultimately deployed on the Ethereum blockchain, the key difference lies in where their deployment infrastructure operates. In off-chain style, the deployment scripts function externally in a centralized environment. Conversely, in on-chain style, proxy instances are created by smart contracts running directly on the blockchain, in a decentralized manner. We discuss the pros and cons of each style in Section 5.8.2.

Observation 5.11) ***There is significantly greater diversity in deployment scenarios for the on-chain style (60.9%) compared to the off-chain style (39.1%), as evidenced by the wider range of use cases.*** When evaluating deployment scenarios and their diversity, a key metric is the ratio of distinct use cases (Section 5.5). Among the 79,171 usage cases identified in our dataset, 48,238 (60.9%) are associated with the on-chain style, while 30,933 (39.1%) correspond to the off-chain style. This distribution indicates that the on-chain style is significantly more prevalent, with a 1.56× higher likelihood of adoption compared to off-chain style. The

dominance of on-chain style suggests its growing popularity in the Ethereum ecosystem, possibly due to its advantages in traceability, reusability, and decentralized orchestration. The two most frequently occurring patterns in terms of use cases are EOA→FA→P, which accounts for 39.81% of all observations, and EOA→P, comprising 39.07%.

Observation 5.12) ***The majority of proxy instances (99.3%) are systematically deployed using the on-chain style.*** A total of 7,191,165 proxy contracts (99.3%) are deployed using the 11 on-chain creational patterns, while only 50,174 proxies (0.69%) are deployed via the off-chain style. Among the 11 on-chain patterns, the second pattern EOA→FA→P and the third pattern EOA→PF→P are the two most widely used in terms of the number of deployed proxy instances. The remaining nine on-chain patterns are essentially specializations of these second and third patterns, typically longer in form. While longer patterns promote greater modularity, reusability, and interoperability across contracts, we observe that simpler patterns tend to dominate in terms of both use case frequency and proxy instance count. Together, the top five on-chain patterns account for 99.26% of all deployed proxy contracts. For each of these top patterns, we systematically identified and analyzed a representative large-scale DApp. Examples showcasing the use of longer on-chain patterns in DApps are discussed in Appendix A.5.

Observation 5.13) ***On-chain creational patterns support larger use cases compared to the off-chain creational pattern.*** Figure 5.14 compares the deployment styles, showing that the CCDF curve for on-chain creational patterns consistently lies above that of the off-chain style. This indicates a higher likelihood of observing larger use cases in the on-chain category. A Mann–Whitney U test confirms that the

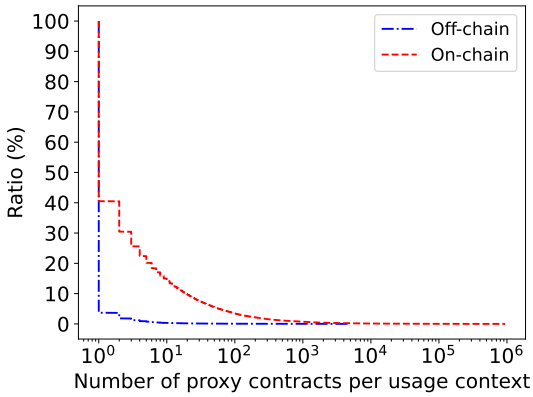


Figure 5.14: CCDF of the number of proxy contracts per use case, comparing on-chain and off-chain deployment styles.

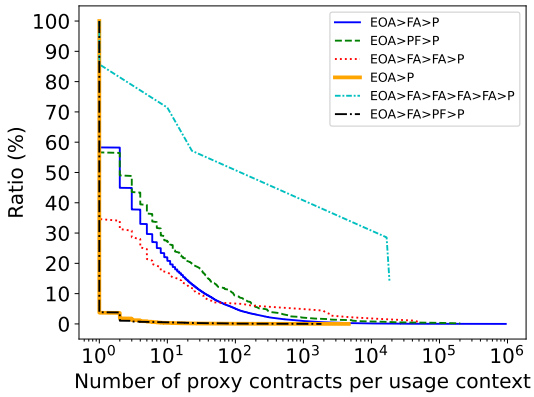


Figure 5.15: CCDF of the number of proxy contracts per use case, comparing the top-5 on-chain creational patterns with the off-chain pattern.

distribution of use case sizes is significantly greater for on-chain patterns than for the off-chain pattern (p-value < 0.05), with Cliff's Delta indicating a medium effect size ($\delta = 0.37$).

Furthermore, Figure 5.15 offers a more detailed comparison of the CCDFs for the top five on-chain creational patterns and the single off-chain pattern. The results show that on-chain patterns generally exhibit a higher probability of larger use cases compared to the off-chain pattern (represented by the red line). Additionally, the CCDF curves of the on-chain patterns decrease more gradually, suggesting that larger use cases, i.e., those involving more proxy contracts, are more frequently observed in on-chain deployments. In contrast, the CCDF curve of the only off-chain pattern (i.e., EOA→P) declines more steeply, indicating a lower probability of encountering large-scale use cases within its distribution.

Observation 5.14) ***On-chain deployment style scales better for non-singleton use cases, while off-chain favors singleton use cases.*** Table 5.3 shows the distribution

of use cases by deployment style (on-chain vs. off-chain) and use case size, measured by the number of proxy contracts per context. When considering the number of proxy contracts (N) per use case, we observe two broad categories: singleton use cases and non-singleton use case. Singleton use cases represent use cases where only a single proxy contract is deployed. These make up the majority, 58,484 use cases (73.9% of the total). Singleton deployments are almost evenly split between the two styles: 28,691 on-chain vs. 29,793 off-chain. This near parity suggests that for small-scale or isolated deployments, both styles are equally viable depending on the DAPP context and constraints, though off-chain deployment slightly leads. Off-chain deployment is likely favored in such cases due to its simplicity and accessibility, often requiring just a script or deployment tool rather than a supporting on-chain contract infrastructure.

In contrast, non-singleton use cases, those with more than one proxy contract, paint a very different picture. Out of the 20,687 such use cases, 19,547 (94.5%) use the on-chain style, while only 1,140 (5.5%) rely on off-chain methods. This stark contrast underscores the critical role of on-chain style in enabling modular, scalable DApps. On-chain deployments are well-suited for DApps that dynamically create multiple proxy instances through contract logic, as seen in factory contracts. These patterns benefit from decentralized traceability, automated and systematic deployment control, advantages that are not easily achievable with off-chain scripts.

To formally evaluate the relationship between deployment style and use case size, we conducted a chi-square test of independence. The results revealed a statistically significant association between the two variables, indicating that the choice of deployment style is not random but systematically related to the scale of the use

Table 5.3: Distribution of use cases by deployment style (on-chain vs. off-chain) and use case size, measured by the number of proxy contracts per use case.

Number of proxy contracts (N)	**Deployment Style**	
	On-chain	*Off-chain*
N > 1 (Non-Singleton)	19,547	1,140
N = 1 (Singleton)	28,691	29,793
Total	48,238	30,933

case. The effect size, measured using Cramér's V, was 0.4, indicating a medium-strength relationship between deployment style and use case size.

Summary

RQ2: What Are the Different Creational Patterns for Deploying Proxy Contracts?

We identified 12 distinct creational patterns for deploying proxy contracts. These patterns fall into two deployment styles: on-chain (11 patterns) and off-chain (1 pattern). On-chain deployment dominates, accounting for 99.3% of all proxy instances and 60.9% of use cases. The most prominent patterns are EOA→FA→P (on-chain) and EOA→P (off-chain). While singleton use cases ($N = 1$) are nearly evenly split between styles, non-singleton use cases ($N > 1$) are overwhelmingly on-chain (94.5%). On-chain deployments are also associated with greater diversity and larger use cases. A chi-square test confirmed a statistically significant relationship between deployment style and use case size, with a medium effect size.

5.7 RQ3: To What Extent Do Proxy Contracts Differ in Type and Adhere to Established Reference Implementations?

5.7.1 Motivation

The widespread adoption of proxy contracts on Ethereum has led to increased interest in understanding their structural variations and usage practices (Salehi et al. 2022; Gnosis 2023; Bodell III et al. 2023). Broadly, proxy contracts can be categorized into two main types: Upgradeable Proxy Contracts (UPCs) and Forwarder Proxy Contracts (Jorge Izquierdo 2018; Salehi et al. 2022). UPCs are primarily used to enable seamless smart contract evolution by allowing the substitution of a logic contract reference without changing the proxy's address, ensuring zero downtime and continued service availability. FPCs, in contrast, delegate calls to other contracts for modularization, permission delegation, or cost-effective cloning of functionality. Despite this conceptual classification, little is known about the current prevalence of these proxy types in the Ethereum ecosystem. This can offer valuable insights into the architectural decisions made by DApp developers, particularly in navigating the tension between immutability and the need for contract evolution. Although earlier work (Salehi et al. 2022) performed some preliminary studies on upgradeability or forwarder types, it was conducted over a limited period (2020–2021) and thus fails to capture long-term trends from an empirical perspective. This short observation window limits its ability to reveal how widely these proxy types have been adopted over time.

Furthermore, A growing number of standardized proxy implementations have emerged to guide best practices in Ethereum smart contract development, (e.g. ERC-897 (Jorge Izquierdo 2018), ERC-1167 (Peter Murray 2018a), etc.). These standards were introduced to address critical concerns such as gas efficiency, upgrade safety, and transparency. However, it remains unclear to what extent developers actually follow these recommended implementations in practice. Given the security risks and operational challenges associated with non-standard or ad hoc proxy designs, evaluating adherence to these reference implementations is essential and can offer insights into industry best practices, reveal gaps in tooling or education, and help guide future improvements in smart contract infrastructure. Motivated by this need, we conduct an early investigation to assess the prevalence of seven reference proxy implementations and examine whether deployed proxy contracts conform to these community-driven specifications.

5.7.2 Approach

Salehi et al. (2022) proposed a preliminary approach approach for detecting the type of proxy contract. However, since no replication package is available, we manually applied their methodology to a statistically representative subset of proxies. Specifically, from the 79,171 detected use case (Section 6), we selected one representative proxy contract per use case. Given that all proxies within a use case are structurally similar, the representative contract was randomly selected. We then drew a statistically representative random sample from these representative contracts, comprising 385 instances, using a 95% confidence level and a 5% margin of error. Our goal was to analyze the distribution of UPCs versus FPCs among these

samples. To ensure consistency in manual classification, we established a rigorous protocol based on the classification procedure described by Salehi et al. (2022). The first author extracted and summarized the relevant steps into a classification guide. Following this protocol, the first and second authors independently applied the method to classify each proxy instance. In cases of disagreement (three in total), the third author facilitated a resolution through discussion. No instances were marked as *unknown*. The classification process required approximately 17 person-hours. For transparency and reproducibility, we include the full sample dataset as part of our replication package.

Furthermore, to operationalize our investigation into the adherence of proxy contracts to community-recommended standards, we employed the *evm-proxy-identification* tool (Gnosis 2023). This tool is designed to detect seven widely used proxy implementations that represent current best practices: four variants based on ERC-897 (Jorge Izquierdo 2018), along with ERC-1167 (Peter Murray 2018a), ERC-1822 (Gabriel Barros 2019b), ERC-1967 (Santiago Palladino 2019b), ERC-1967 (Beacon) (Santiago Palladino 2019b), OpenZeppelin's Proxy Pattern (OpenZeppelin 2018a), and GnosisSafeProxy[9]. All of these standards, except ERC-1167, define a unique storage slot to hold the logic contract address, which enables reliable detection through storage inspection. ERC-1167 proxies, in contrast, are identified based on their distinctive bytecode pattern.Due to API request quotas, we applied this detection tool to a statistically representative sample of 13,776 proxy contracts, selected using a 99% confidence level and a 1% margin of error. Each contract in the sample was classified into one of the seven reference implementations. For contracts that did not match any known pattern, we introduced an additional category

[9] https://github.com/safe-global/safe-contracts

labeled “Customized”, enabling us to capture non-standard or bespoke designs. This analysis provides a first step toward understanding how widely developers adopt established proxy standards and where deviations may signal gaps in tooling, documentation, or awareness. The full sample dataset and results are included in our replication package.

5.7.3 Findings

Observation 5.15) ***Just above two-thirds of usage cases employ FPCs, whereas 32.2% use UPCs for the purpose of contract evolution*** Out of 385 representative proxy contracts, 124 (32.2%) were designed for contract evolution (UPCs), while the remaining 261 (67.8%) served solely as forwarder proxy contracts (FPCs). As emphasized in the introduction, upgradeability proxies play a key role in enabling the long-term evolution of decentralized applications, but they also introduce critical risks that require continuous monitoring. These findings highlight the need for reliable tooling capable of distinguishing between UPCs and FPCs, as well as tracking logic contract replacements and changes to proxy administrators, operations that, if left unchecked, can compromise the security and governance of deployed applications.

Observation 5.16) ***Just below 30% of use caeses employed ERC-1167 minimal proxies, while around 21% used customized implementations.*** According to the results of the *evm-proxy-identification* tool, shown in Figure 5.16, a substantial majority of use cases (approximately 79%) adhere to established best practices when adopting proxy contracts. Among these, the ERC-1167 minimal proxy implementation is the most prevalent, accounting for roughly 29.38% of cases, followed by

ERC-897 (25.08%) and ERC-1967 (22.91%). The popularity of ERC-1167 is likely due to its significant impact on reducing deployment costs when cloning contract functionality (Peter Murray 2018a). Based on this, we conjecture that a large proportion of proxies in Ethereum are primarily adopted for cost-efficient deployment rather than contract evolution.

Notably, around 21.1% of the use cases adopted "Customized" proxy implementations, indicating that they do not conform to any of the seven studied standards. Several factors may explain this outcome. First, these proxies may represent ad hoc or project-specific implementations that intentionally deviate from standard patterns, either to meet specialized requirements or to obscure their behavior. Second, they may originate from older or legacy systems deployed before the emergence of current standards. Third, some developers may simply be unaware of available proxy standards or choose not to follow them due to limited tooling or documentation. Additionally, this classification may be partially influenced by limitations of the *evm-proxy-identification* tool itself. The tool relies on specific bytecode patterns and well-defined storage slot conventions to identify standard implementations. Customized proxies that slightly modify these conventions e.g., by changing storage slot layouts or proxy logic structure, could evade detection even if they are derived from standard patterns. As such, the reported 21.1% may include false negatives. We discuss these possibilities in more detail in Section 5.9. The least adopted implementations were ERC-1967 (Beacon), ERC-1822, and Gnosis Safe Proxy, with adoption rates of just 0.28%, 0.25%, and 0.13%, respectively.

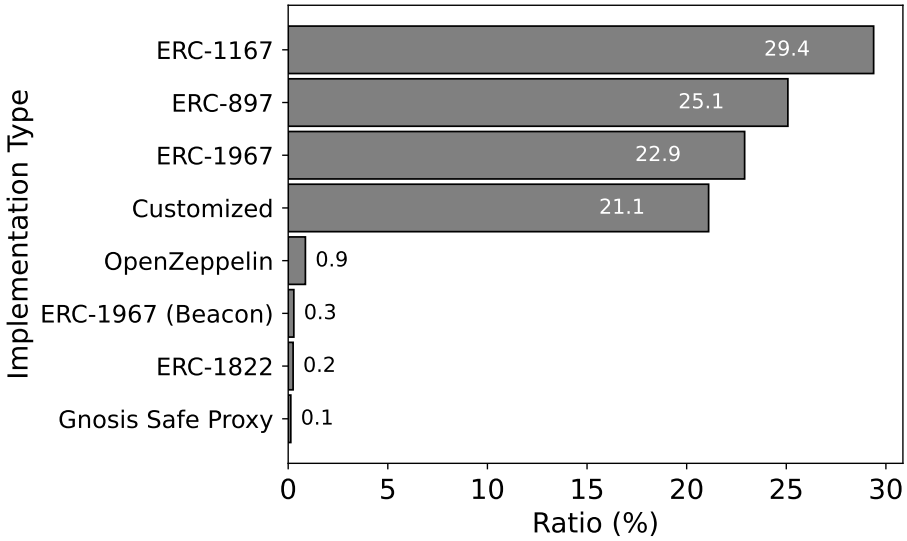


Figure 5.16: Distribution of proxy use cases by reference implementation type.

Summary

RQ3: To What Extent Do Proxy Contracts Differ in Type and Adhere to Established Reference Implementations?

We observe a predominant use of forwarder proxy contracts (67.8%) compared to upgradeability proxies (32.2%), underscoring the need for effective monitoring tools to ensure secure contract evolution and governance. Furthermore, while the majority of proxies follow established standards, most notably ERC-1167 (29.4%), approximately 21.1% are implemented using customized, non-standard designs, revealing considerable diversity in proxy usage and potential gaps in tooling or developer awareness.

5.8 Discussion

5.8.1 Trade-offs of the Proxy Pattern in Smart Contract

In this section, we examine the key trade-offs introduced by the proxy pattern in smart contract design, highlighting its benefits for flexibility and contract evolution alongside its architectural, security, and transparency challenges.

5.8.1.1 Advantages

- **Contract Evolution**. The proxy pattern enable seamless contract evolution in decentralized applications (DApps). By decoupling the contract's logic from its state, where the proxy contract retains the persistent storage, developers can upgrade the underlying logic contract without altering the contract's public address. This ensures that users and other contracts can continue interacting with the same proxy address, while the actual code being executed can be transparently updated or evolved behind the scenes. This design eliminates the need for user or data migration, which is often error-prone, costly, and disruptive. In order contract evolution mechanisms, migrating users to a new contract instance may require reinitialization of state, manual intervention, or re-establishing trust and integrations, each of which introduces friction and potential failure points. This pattern remove these risks by enabling contract evolution without changing the interface address or relocating data, thereby preserving user continuity, service availability, and on-chain interoperability. As such, they are especially valuable for maintaining long-lived DApps that require ongoing iteration, bug fixes, and feature expansion.

- **Modularity**. The proxy pattern enhances modularity in smart contract design by allowing developers to break down complex systems into smaller, independent logic contracts. Instead of placing all functionality into a single monolithic contract, developers can structure decentralized applications (DApps) as a collection of modular contracts, each responsible for a specific feature or domain. The proxy contract serves as a central dispatcher that routes function calls to the appropriate logic contracts. This modular architecture is particularly useful in overcoming the EVM's maximum contract size limit of 24KB, which can restrict the deployment of feature-rich applications. By distributing functionality across multiple contracts and managing them through a proxy, developers can build large-scale, well-structured systems while maintaining a clean separation of concerns. Standards such as ERC-2535 (Mudge 2020a) demonstrate how the proxy pattern can support a flexible delegation mechanism, enabling the composition of contract functionality across multiple specialized components.

- **Cost-Efficient Deployment**. The proxy pattern helps reduce deployment costs by allowing developers to share a single logic contract across multiple contract instances. This is made possible through the use of the `delegatecall` instruction, which executes the logic contract in the context of the proxy's storage. Since the proxy holds the persistent state, there is no need to duplicate the logic contract for each deployment. Instead, only lightweight proxy contracts, with minimal delegation code, need to be deployed to cheaply clone the logic contract's functionality. Standards such as ERC-1167 (Peter Murray 2018a) formalize this approach through minimal proxies, making it highly efficient

in terms of gas usage. This approach becomes particularly important in scenarios where a rich and complex contract, such as a multi-signature wallet, personal vault, or module-based account, is required to be deployed for every individual user. For instance, in the Gnosis Safe architecture, each user receives their own wallet instance, but rather than deploying the full logic code repeatedly, Gnosis deploys lightweight proxies that delegate to a single master copy. Without this pattern, deploying such feature-rich contracts for thousands or millions of users would result in prohibitively high gas costs.

- **Third-Party Contract Reuse**. The proxy pattern enables developers to reuse an already deployed contract (e.g., the Gnosis Safe wallet contract[10]) by deploying a proxy that forwards calls to the target contract using `delegatecall`. This allows the proxy to replicate the behavior of the target contract while maintaining its own storage. Such reuse brings several benefits. First, it significantly reduces development and deployment overhead, as there is no need to rewrite or redeploy well-tested and widely used contracts. Second, it allows developers to inherit the trust and auditability of battle-tested implementations, improving security and reliability. Third, it promotes standardization and interoperability within the Ethereum ecosystem, as multiple projects can integrate with and extend the same underlying logic in a consistent manner. However, this approach requires careful alignment of storage layouts and a clear understanding of the original contract's permission model, as access controls may limit certain interactions or introduce integration risks.

[10] https://etherscan.io/address/0xd9db270c1b5e3bd161e8c8503c55ceabee709552

- **Access Control**. A key benefit of the proxy pattern lies in its ability to enforce strict access control over contract behavior. By decoupling execution logic from the proxy's state and interface, developers can design systems where only specific entities are authorized to trigger delegated calls. This reduces the risk of unintended interactions, unauthorized access, or exposure of sensitive functions to external users. An illustrative example of this principle is the puppet proxy pattern, formalized in ERC-7613 (Żuk 2024) In this design, the proxy contract does not expose any public API and remains inert to all callers except its original deployer. Only the deployer can invoke functionality by passing a target address and delegation payload, while all other calls are ignored. This ensures that only the designated controller can execute logic on behalf of the proxy, effectively minimizing the attack surface.

5.8.1.2 Disadvantages

- **Architectural Complexity**. The proxy pattern adds an extra layer of indirection to DApp design, increasing both conceptual and operational complexity. Rather than interacting with a single contract, developers must manage a system composed of at least two interdependent components: the proxy and the logic (or implementation) contract. This separation introduces new challenges in terms of deployment, coordination, and version management. Developers must ensure that the proxy and logic contracts remain compatible across upgrades and that critical elements such as storage layouts are correctly aligned to prevent corruption. Additionally, the indirection caused by

`delegatecall` can make debugging and auditing more difficult, as the execution context is decoupled from the codebase. These factors not only increase the cognitive overhead for developers but also raise the likelihood of subtle bugs and integration issues in complex systems.

- **Gas Overhead**. While the proxy pattern can reduce deployment costs, especially through patterns like ERC-1167 minimal proxies, their use often introduces runtime gas overhead due to the additional indirection layer. Each interaction with a proxy involves a `delegatecall`, which has a base gas cost of 700, along with additional costs for memory expansion and data handling. In practice, this means that every function call incurs extra execution steps compared to calling a standalone contract directly. This overhead may be negligible in many applications, but for gas-sensitive use cases or high-frequency interactions (e.g., DeFi protocols), it can accumulate significantly over time. As a result, developers must weigh the benefits of the proxy pattern against the long-term operational cost introduced by these indirection mechanisms.

- **Centralization Risks**. The proxy pattern can introduce centralization risks when contract evolution is controlled by a single privileged entity. Without proper access controls and transparent governance mechanisms, the upgrade authority may unilaterally introduce malicious changes, siphon assets, or modify core functionality, actions that undermine user trust and protocol integrity. Even in well-intentioned settings, the lack of visibility into upgrade processes can raise security concerns and reduce confidence among users and

stakeholders. Conversely, in decentralized systems, coordinating contract evolution among distributed actors, such as DAO members or governance token holders, can be slow, complex, and contentious. This inherent tension between flexibility in contract contract evolutions and decentralized control makes proxy-based governance a challenging aspect to manage effectively, especially for high-value or widely used smart contracts.

- **New Attack Vectors**. While proxy contracts bring certain benefits, they also introduce a range of security risks that must be carefully mitigated. Below are some of the most common and impactful vulnerabilities observed in proxy-based systems:
  - **Admin Key Compromise**. If the private key controlling the proxy's admin is compromised, an attacker can unilaterally upgrade the proxy to point to a malicious implementation. This kind of attack targets the proxy's governance authority itself. In the PAID Network exploit (Behnke 2021), the attacker gained access to the admin key and replaced the implementation, enabling unauthorized minting and token theft.
  - **Uninitialized Proxy**. If a proxy is deployed without calling its initializer, a malicious actor may invoke it and gain ownership or control over upgrade functionality. In February 2022, the Wormhole core bridge contract fell victim to this vulnerability. A whitehat discovered that the uninitialized implementation could be initialized by anyone, which would have enabled them to self-destruct it, potentially locking up $1.8 billion of user funds (Chainalysis Team 2022).

- **Storage Collisions**: Improper alignment between the proxy's and logic contract's storage layouts can overwrite critical data, such as ownership variables or balances. In the Audius governance contract exploit (Audius Blog 2022), misaligned storage allowed an attacker to take control and transfer $6M worth of AUDIO tokens.
- **Function Selector Clashes**. In EVM, each function is identified by a 4-byte selector derived from its signature. When two different functions across the proxy and its implementations produce the same selector, a clash occurs. This may cause the proxy to call its own internal function instead of forwarding to the correct logic, leading to incorrect or unexpected behavior. This vulnerability becomes particularly dangerous when attackers deliberately craft function signatures to mimic internal proxy functions (e.g., `proxyOwner()`) and hide malicious behavior behind seemingly benign function names. Such a selector clash was shown to allow a concealed backdoor, enabling unauthorized logic bypass and fund access. Selector collisions are difficult to detect because they must examine both the proxy and all associated logic contracts simultaneously to identify overlapping function selectors (Palladino 2018).
- **Insecure Governance Mechanism**. Proxy contracts rely on governance to define who can evolve a contract, under what conditions, and through which procedures. When upgrade functions, are left publicly accessible or lack proper access controls, any user may gain the ability to alter the proxy's logic. In the Audius governance exploit (Audius Blog 2022), the attacker exploited a misconfigured upgrade mechanism to replace the

implementation contract and steal over $6 million in tokens. Similar flaws have been identified in audits, where missing access modifiers or overly permissive governance led to unauthorized control over contract evolution.

- **Transparency Concerns**. Proxy contracts can obscure the true behavior of a decentralized application, as end-users may not be aware that they are interacting with a proxy rather than a standalone contract. This can lead to confusion or misinterpretation of the contract's functionality, particularly if the underlying logic contract is evolved without clear communication or on-chain disclosure. The separation between interface and logic complicates auditing, verification, and trust. We further discuss these implications in Section 5.8.1.2and Section 5.8.3.

5.8.2 Trade-offs of Off-chain vs. On-chain Deployment Styles

In this section, we compare off-chain and on-chain deployment styles for proxy contracts across four key themes: i) Development Workflow & Usability, ii) Deployment Architecture & Patterns, iii) System-Level Properties, and iv) Governance, Transparency & Observability.

5.8.2.1 Development Workflow & Usability

This theme captures concerns related to the developer experience, deployment efficiency, collaboration, and lifecycle management.

- **Ease of Deployment**. Off-chain deployment relies on manually authored scripts or CLI tools, which must be executed each time a proxy is deployed. This approach is straightforward but repetitive. In contrast, on-chain deployment abstracts this logic into a factory contract, enabling one-click deployment via function calls.

- **Deployment Automation & Reuse**. Off-chain scripts are typically custom and hard to generalize or reuse across projects. On-chain factories encode deployment logic once and support high levels of reuse, making them ideal for automated, large-scale deployments.

- **Contract Evolution**. In off-chain setups, coordinating contract evolution across a large number of deployed proxy contracts requires manual tracking of each proxy, logic contract pair, often relying on custom scripts and ad hoc processes. This approach is time-consuming, error-prone, and difficult to audit. In contrast, on-chain factories can maintain an internal registry of deployed proxies and their associated logic contracts, allowing contract evolution to be streamlined across all instances. In scenarios involving mass upgrades, the factory can efficiently update the logic address for multiple proxies in a controlled and auditable manner, greatly reducing coordination overhead and risk.

- **Multi-user**. Off-chain deployments are typically managed by a single operator or script, resulting in a centralized workflow tied to a specific environment. While this works for small teams, it becomes limiting as projects scale or involve multiple contributors. On-chain factory contracts provide a shared deployment interface accessible to authorized users, enabling consistent and

secure deployments across teams or communities. Factories can enforce permissioning (e.g., role-based access), promoting collaboration, decentralization, and eliminating single points of control.

In summery, off-chain deployment is simpler and faster for small teams or isolated use cases but becomes difficult to scale or coordinate as projects grow. On-chain deployment introduces initial complexity but provides significantly better reusability, maintainability, and team collaboration in the long term.

5.8.2.2 Deployment Architecture & Patterns

This theme addresses the structural organization and design sophistication of deployment mechanisms, including how modular, predictable, and reusable they are.

- **Architectural Complexity**. Off-chain deployment is architecturally straightforward, relying on scripts and manual execution without the need for additional on-chain components. In contrast, on-chain deployment requires developing and maintaining additional smart contracts, such as factories and proxies, which introduces more design effort and code complexity. However, this added complexity enables greater modularity, automation, and scalability.

- **Standardization**. Off-chain approaches are often ad hoc, with scripts tailored to individual projects, leading to inconsistency and poor reusability. On-chain deployments typically follow well-defined standards (e.g., OpenZeppelin factories), improving maintainability and cross-project interoperability.

- **Predictability of Address**. Off-chain deployments rely on the deployer's address and transaction nonce, making the resulting proxy contract addresses

difficult to predict ahead of time. This unpredictability poses challenges in scenarios where the contract address needs to be referenced or integrated before deployment. In contrast, on-chain deployments can leverage the `CREATE2` EVM instruction, which allows deterministic computation of contract addresses based on known inputs such as the factory address, a salt value, and the contract's init code. This address predictability is especially useful for enabling meta-transactions (where users authorize interactions with contracts before they are deployed), pre-registration or whitelisting in external contracts, and off-chain integrations that rely on knowing the target address in advance. As a result, predictable addressing enhances planning, automation, and interoperability in decentralized applications.

- **Batch Deployment**. Batch deployment refers to the ability to create multiple proxy contracts within a single blockchain transaction, rather than sending individual transactions for each instance. In off-chain setups, achieving this typically requires custom scripting to manage and sequence multiple deployments, which can be inefficient, error-prone, and harder to audit. In contrast, on-chain factories can natively support batch creation using loop-based or multi-call functions, allowing all contracts to be deployed atomically in one transaction. This approach improves deployment efficiency, ensures consistency, and simplifies failure handling. Batch deployment is especially valuable in scenarios like mass NFT minting, multi-user contract initialization, or registry setup, where speed, cost-effectiveness, and operational reliability are critical.

In summary, off-chain deployment offers architectural simplicity and flexibility through script-driven patterns but lacks consistency and composability. On-chain deployment introduces greater design complexity through factories and proxy standards but enables higher modularity, predictable address generation, and scalable patterns like batch deployment, making it more suitable for standardized, reusable deployment workflows.

5.8.2.3 System-Level Properties

This theme addresses the operational characteristics of deployment styles, particularly their impact on scalability, efficiency, and system integration.

- **Scalability** Off-chain deployments are not well-suited for large-scale applications, as each proxy must be deployed individually through scripts or automation tooling. This adds coordination overhead and increases the risk of failure in complex deployment pipelines. In contrast, on-chain deployment scales naturally through factory contracts which can dynamically create new proxies on demand as part of the application's runtime behavior, enabling more flexible and automated deployment at scale.

- **Gas Efficiency**. Off-chain deployments typically have lower gas costs for small-scale or one-time deployments, as they avoid the overhead of deploying and interacting with on-chain factory contracts. However, for large-scale or repeated deployments, on-chain systems become more gas-efficient over time. The initial factory deployment cost is amortized across many proxies,

and ERC-1167 minimal proxy patterns (Peter Murray 2018a) significantly reduce per-instance gas consumption. Additionally, on-chain factories can optimize batch deployment and reuse logic, making them more cost-effective in high-volume or long-lived applications.

- **Interoperability**. Off-chain deployment logic is usually project-specific and not accessible or usable by other smart contracts. On-chain factories, by contrast, expose standardized, callable interfaces that other contracts and external systems can interact with. This makes on-chain deployment more composable and better suited for integration into broader DApp ecosystems or protocol-level workflows.

In summary, while off-chain deployment may offer short-term efficiency for isolated use cases, it does not scale well and lacks integration capabilities. On-chain deployment, although initially more gas-intensive, provides long-term advantages through scalable factory contracts, gas amortization using minimal proxies, and improved interoperability with other smart contracts and DApps.

5.8.2.4 Governance, Transparency & Observability

This theme evaluates how open, secure, and verifiable the deployment process is from the perspective of external users, auditors, and automated tools.

- **Decentralization**. Off-chain deployment relies on centralized scripts executed by a single operator or CI/CD system, concentrating control and limiting trust guarantees. In contrast, on-chain deployments operate through smart contract factories, which once deployed, allow anyone to interact with the same

verifiable deployment logic. This decentralization ensures that deployment processes are no longer bottlenecked by a single party and are transparently governed by code.

- **Auditability**. Because off-chain scripts are not stored or executed on the blockchain, they cannot be audited or verified after the fact. Any mistakes, misconfigurations, or changes in behavior are difficult to trace. On the other hand, on-chain deployment logic is fully visible and immutable once deployed. This enables third parties, auditors, community members, or users, to inspect and verify how contracts are deployed, improving trust and reducing attack surfaces.

- **Indexability (Events)**. Off-chain deployments often fail to emit standardized on-chain events, making it hard to track contract creation or deployment metadata. On-chain factories typically emit structured events like `ProxyCreated` or `ImplementationUpdated`, which are easily indexable by block explorers, analytics tools, and monitoring systems. This improves observability, integration, and system-wide transparency (Wood 2023).

In summary, off-chain deployment lacks transparency, decentralization, and verifiability, limiting its trustworthiness and integration with ecosystem tooling. On-chain deployment ensures that deployment logic and events are publicly visible, verifiable, and indexable, thereby enhancing auditability, community trust, and system-wide observability.

5.8.3 Implications

Implication 5.1) ***Research should adopt a system-level perspective when analyzing smart contracts.*** Our motivation study (Section 5.3) revealed that smart contracts are increasingly interdependent, with many exhibiting high degrees of coupling with other contracts. This shift marks a transition in smart contract architecture, from standalone, monolithic designs to modular and interconnected systems. Accordingly, researchers should move beyond analyzing individual contracts in isolation and instead adopt a system-level perspective that considers the broader context of decentralized applications (DApps) and the ecosystem of contracts they comprise. Despite this trend, there is currently no systematic effort to define or delineate the boundaries of such systems on Ethereum. Future research should aim to establish rigorous methods for identifying and characterizing smart contract systems, including their scope, composition, and dependencies.

In the case of proxy contracts, our findings demonstrate that they introduce an additional layer of indirection and have played a pivotal role in smart contract interactions since Ethereum's inception (Observation 8). To facilitate system-level analysis, we proposed the concept of a "use case", a grouping of proxy contracts deployed by the same entity and serving a similar functional purpose, forming coherent clusters. This concept serves as a foundation for our analysis and findings in RQ2 and RQ3. Additionally, we identified large-scale decentralized systems that utilize the top-5 on-chain proxy creational patterns (Appendix A.5). Nevertheless, further investigation is needed to comprehensively understand the architecture and operational dynamics of the broader ecosystem of 7.2 million detected proxies.

Such insights are critical for improving smart contract analysis tools, informing governance practices, and guiding best practices for modular contract design.

Implication 5.2) ***Proxy contracts reduce transparency in DApps.*** Proxy contracts can reduce transparency in decentralized applications by introducing an additional layer of indirection between users and the underlying logic contracts (Section 5.8.1.2). While proxy contracts offer architectural benefits, they becomes problematic when essential metadata, such as the source code or address of the logic contract, is hidden or missing.

To validate this concern, we conducted a post-hoc study to examine how proxy contracts are used in practice within DApps. We compiled a dataset of 3,767 Ethereum DApps by aggregating information from three major marketplaces: stateofthedapps.com, dappradar.com, and dapp.com. By cross-referencing the smart contracts associated with these DApps against our proxy detection dataset (Section 5.5), we found that 307 DApps (8.14%) included at least one proxy contract.

A closer inspection of these 307 DApps revealed that 297 (97.1%) failed to publish some or all of their corresponding logic contract addresses. This lack of disclosure significantly undermines transparency, making it difficult for users, auditors, and fellow developers to verify the behavior or interact with the underlying logic. Several factors could contribute to this practice: a desire to shield logic contracts from direct attack, the intention to preserve proprietary logic, or a lack of awareness regarding the importance of transparency. In the case of UPCs, frequent changes to logic contracts may also deter developers from maintaining up-to-date list of logic contracts.

Furthermore, our study revealed that over 20% of use cases involve proxies that deviate from the storage layout conventions used by popular reference implementations (Section 16). This design choice impairs the ability of tools like Etherscan to trace and display the active logic contract, thereby exacerbating the transparency issue.

These findings highlight a critical gap: without visibility into logic contracts, auditing becomes harder, interoperability is hindered, and the completeness of DApp records in marketplaces suffers. To address this, we advocate that DApp developers consistently publish both proxy and logic contract addresses. Marketplaces can also integrate services such as Etherscan Proxy Verification to automate logic contract resolution. Additionally, our proxy detection method (Section 5.4.1) offers a practical accurate solution for identifying and associating proxy contracts with their logic contracts, helping maintain accurate and transparent registries of DApp components.

Implication 5.3) ***Developers should weigh the benefits and challenges of the proxy pattern and choose the deployment styles that best meet the needs of their DApps.*** The decision flowchart in Figure 5.17 offers a structured guide to help developers choose between off-chain and on-chain styles for deploying proxy contracts. It is grounded in the trade-offs discussed in Section 5.8.2 and operationalizes key concerns such as scalability, auditability, maintainability, and decentralization. This model supports and extends the empirical findings in Observation 14, which reveal a strong correlation between deployment style and use case size. Specifically, it was observed that on-chain deployment dominates in non-singleton use

cases (those with multiple proxies), while off-chain deployment style is slightly more common in singleton settings.

The flowchart begins by distinguishing between singleton and non-singleton use cases. Singleton use cases are defined as scenarios where a DApp requires only one or a small number of proxy contracts, each serving a distinct purpose. We observed that use cases are nearly evenly split between off-chain and on-chain deployment styles, suggesting that for isolated or small-scale deployments, both approaches are being used in practice of viable depending on the constraints and requirements. In contrast, non-singleton use cases, where a DApp needs to deploy many identical proxy–logic contract pairs dynamically, either in batch or on-demand, strongly favor the on-chain style due to its scalability and automation capabilities (Observation 14).

More specifically, from the singleton branch, the next decision point asks whether predictability of the proxy contract address is important. If so, developers are directed toward on-chain deployment using `CREATE2`, which allows precomputing the proxy contract address based on known parameters. This is especially useful in scenarios requiring pre-registration or integration with other systems before deployment (Section 5.8.2.2).

If predictability is not required, the flowchart considers whether multiple users or other smart contracts need to initiate or manage deployments. This addresses the need for shared control, permissioning, and collaborative deployment. On-chain factories provide a secure and verifiable way to support such multi-party or programmatic deployment logic, aligning with governance and decentralization goals (Section 5.8.2.1, and 5.8.2.3).

When neither predictability nor coordination is necessary, the chart next evaluates the need for auditability and deployment transparency. Developers who prioritize traceability and public verification of deployment actions, especially in decentralized or user-facing systems, are advised to use on-chain factories that emit standardize events and retain immutable logic. If these criteria are not important, then the simplest, cheapest and most direct method, off-chain deployment via an EOA, is appropriate (Section 5.8.2.4).

For non-singleton use cases, the chart focuses on whether there is a need to evolve the deployment logic over time. If so, an on-chain upgradeable proxy factory is recommended, enabling changes to how deployments are performed while maintaining consistency and verifiability. upgradeable proxy factory is a specific type of factory that is evolvable through the proxy pattern evolution mechanism. If not, a standard on-chain factory using `CREATE2` or `CREATE` (where `CREATE2` supports address predictability and `CREATE` is a cheaper but non-deterministic alternative) is sufficient, offering batch deployment, modularity, and composability.

In essence, the flowchart synthesizes the architectural, operational, and governance trade-offs into a structured decision-making process. It serves both as a tool for practitioner guidance and as a conceptual bridge linking empirical findings with system-level design considerations. It reinforces that off-chain deployment remains effective for small-scale, isolated scenarios, while on-chain deployment is indispensable for scalable, automated, and collaborative contract systems.

Implication 5.4) ***Future studies should delve into the release engineering process of smart contracts that evolve through the proxy pattern.*** Chen et al. (2021a) investigated why developers terminate smart contracts, using a method based on code

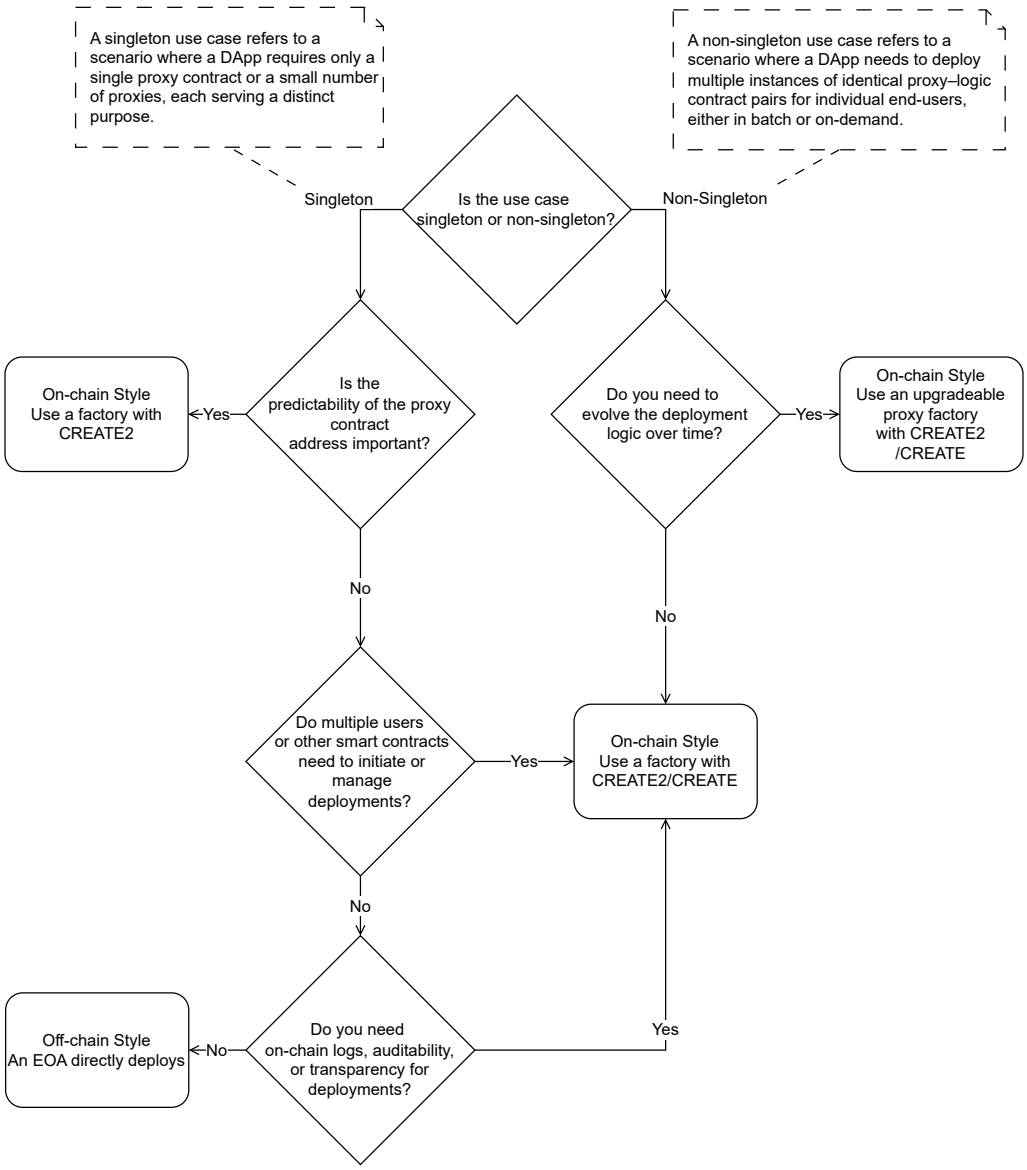


Figure 5.17: Decision flowchart for selecting between off-chain and on-chain deployment styles.

similarity to detect self-destructed contracts and trace their subsequent versions. Solidity's `SelfDestruct` operation permanently removes a contract and transfers its remaining balance to a designated address. While informative, improper use of this feature can result in critical vulnerabilities or irreversible fund losses. Although Chen et al. (2021a) pioneered efforts to understand contract release lifecycles, their study did not examine release engineering through the proxy pattern contract evolution mechanism.

Wöhrer and Zdun (2021) explored how DevOps practices apply to blockchain software development, particularly Ethereum smart contracts. They outlined a DevOps model covering continuous integration and delivery, highlighting the unique challenges posed by the immutability of deployed contracts. While this work contributed foundational insights into blockchain DevOps, it did not focus on proxy-based release strategies, leaving an important gap in our understanding of how smart contracts evolve in practice.

Notably, Chen et al. (2021a) identified proxy contracts as a central mechanism for enabling smart contract evolution and emphasized the lack of large-scale empirical data to systematically study this pattern on Ethereum. To address this gap, we developed a method to identify an accurate and comprehensive dataset of proxy contracts along with their associated logic contracts, laying the groundwork for future research. As demonstrated in our manual investigation, over 32% of use cases involve Upgradeable Proxy Contracts (Observation 15), highlighting

their widespread adoption. However, the concrete mechanisms, evolutionary patterns, and governance practices underpinning this form of contract evolution remain largely unexplored. To encourage deeper investigation into proxy-based release engineering, we propose six guiding questions to frame future research in this domain:

- Q1. How prevalent are different types of proxy contracts? While our manual analysis on a representative sample (Section 4.6) provided initial insights into the prevalence of different types of proxy contracts (Observation 15), there has been no comprehensive, large-scale, and fully automated study on the prevalence of Upgradeable Proxy Contracts (UPCs) versus Forwarder Proxy Contracts (FPCs) since Ethereum's inception. Conducting such an empirical study entails three critical subtasks: (i) detecting whether a given contract is a proxy, (ii) classifying the identified proxy as either a UPC or FPC, and (iii) determining the specific variant of the UPC, such as Transparent, ERC-1882 Universal, or Beacon UPC patterns (Section 2.15.6). These tasks are especially challenging due to the high percentage of closed-source contracts on Ethereum, up to 99%, which necessitates accurate bytecode-level detection methods that do not rely on source code availability. However, current tooling is inadequate (Section 3.3.2). Some approaches rely on source code (Bodell III et al. 2023), while others use heuristic-driven strategies (Qasse et al. 2024). To date, only one method (Salehi et al. 2022) attempts bytecode-level detection of proxy contracts. However, it is closed-source, supports only a limited subset of three variants and has not been empirically evaluated, leaving its practical effectiveness uncertain (Salehi et al. 2022). Moreover, we found that approximately

21% of proxy contracts could not be resolved by the `evm-proxy-detection` tool, underscoring the limitations of existing static analysis methods. This highlights a pressing need to improve detection tooling, particularly at the bytecode level, to enable accurate, scalable classification of proxy contract types and variants across the Ethereum ecosystem. The following five research questions directly depend on the availability of a reliable and comprehensive detection mechanism for identifying upgradeable proxy contracts, mechanisms that enable contract evolution and allow the reconstruction of smart contract release histories.

- Q2. How many DApps use UPCs versus FPCs in practice? As a preliminary investigation, we conducted a post hoc, small-scale manual study on the top 50 most active DApps to shed light on the practical application of the proxy pattern (Appendix A.2). Our findings revealed that 86% of these DApps employ proxy contracts to facilitate contract evolution. While this offers a promising preliminary indication of the practical application of the proxy pattern in practice, a more comprehensive, automated study is needed to systematically quantify the use of Upgradeable Proxy Contracts (UPCs) versus Forwarder Proxy Contracts (FPCs) across a broader range of DApps.

- Q3. What is the distribution of release frequency in upgradeable DApps? In other words, how often is the logic contract evolved, and what does this reveal about contract evolution trends in Ethereum? Understanding this frequency can shed light on how actively developers maintain and improve their deployed systems. Moreover, it enables a comparison between proxy-based evolution and other mechanisms such as `selfdestruct`. Such an analysis can also be

contrasted with findings from prior studies (Chen et al. 2021a), helping to contextualize the evolution dynamics enabled by the proxy pattern within the broader landscape of smart contract lifecycle management.

- Q4. What is the typical duration between successive contract evolutions in upgradeable DApps, and how does it differ from traditional software release cycles? Unlike traditional software systems where release cycles are often scheduled, flexible, and reversible through patches or rollbacks, smart contracts operate under strict immutability and higher deployment costs. As a result, upgradeable DApps face unique trade-offs: contract evolution must be carefully planned, auditable, and typically irreversible once executed. Analyzing the intervals between successive upgrades in upgradeable DApps can provide insights into whether blockchain developers adopt agile, iterative practices or lean toward more conservative and infrequent release patterns due to operational risks and trust considerations.

- Q5. What techniques are used to ensure governance in upgradeable DApps? Since smart contract evolution inherently challenges the principle of immutability, upgrades must be carried out in a secure, controlled, and transparent manner. This requires well-defined governance mechanisms, rules and processes that oversee how upgrades are proposed, approved, and enacted. Commonly used governance techniques include Externally Owned Accounts (EOAs), multi-signature (multi-sig) wallets, timelocks, pausable contracts, escape hatches, commit-reveal schemes, and on-chain or off-chain voting (Liu et al. 2023b; OpenZeppelin 2023a; Palladino 2020). Understanding how developers implement and combine these mechanisms is critical to assessing their effectiveness

in maintaining user trust, particularly when control over contract behavior can change after deployment.

- Q6. To what extent is a suitable governance technique used that aligns with the size of a DApp's user base? As the number of users increases, the stakes and social risks associated with contract evolution will likely also grow. For example, if a DApp with thousands of users relies on a single EOA to authorize changes, this indicates poor governance and exposes the system to potential misuse or centralization. Therefore, it is critical to assess whether governance mechanisms scale appropriately with the DApp's user base. Studying this relationship can offer insights into whether developers prioritize robust, decentralized governance models in proportion to their application's reach and criticality.

Addressing these queries demands automated tools for spotting proxy contracts, their types, and determining governance methods. Some research has begun in this direction (Salehi et al. 2022; Bodell III et al. 2023; Qasse et al. 2024), yet there is ample room to further explore the breadth and depth of release engineering in Ethereum.

5.9 Threats to Validity

In this section, we discuss potential threats to the validity of our findings, covering construct, internal, and external validity dimensions.

5.9.1 Construct Validity

Detecting proxies is not a trivial task. Similar to Salehi et al. (2022), we proposed an approach that identifies proxies based on their runtime characteristics. However, unlike Salehi et al. (2022), whose method relies on transaction replay and focuses on proxies implemented via the `fallback` function, our approach improves upon this by leveraging historical transactions and detecting proxies implemented beyond the typical `fallback` pattern.

Since Salehi et al. (2022) did not provide a replication package, we were unable to directly compare our method with theirs. Nevertheless, we tested and evaluated our approach using a carefully constructed ground truth dataset (Section 5.4.2) and demonstrated that it can detect a broader range of proxy contracts than prior work. However, the process of labeling this ground truth manually may introduce a potential source of bias or human error. To mitigate this, multiple annotators cross-checked labels and ambiguous cases were resolved through discussion.

We acknowledge that, while a behavior-based detection method is effective, it cannot detect inactive proxies, i.e., proxy contracts whose proxy functionality is never exercised. In our evaluation (Section 5.4.2), we found that 28 (7.27%) of the representative sample of deployed contracts were inactive proxies. While these contracts are not necessarily toy examples or impractical, our approach could still identify them if applied periodically over time, once they become active. To ensure transparency, we retained inactive proxies in our ground truth and reported evaluation metrics separately for active and inactive cases to avoid construct validity issues.

To distinguish between different proxy reference implementations in RQ3, we used the *evm-proxy-detection* tool developed by Gnosis[11], a reputable open-source Ethereum project. This tool classifies proxies based on their bytecode or storage layout. It successfully identified the reference implementation for 79% of the proxies in our representative sample (Observation 16), offering meaningful insight into the prevalence of known implementations. The remaining 21% of proxies were not matched to any known reference implementation. However, this does not imply flawed design. Many proxies predate widely adopted standards (e.g., post-2018), as shown in Figure 5.8 (Observation 7). Upon further analysis of these "Customized" cases, we found that minor variations in bytecode, especially among minimal proxies, prevented the tool from classifying them. Some developers also adopt custom layouts that diverge slightly from standard templates, making automated detection difficult. We suggest that future work extend the *evm-proxy-detection* tool to support more reference implementations. Doing so would enhance the coverage of classification tools and help replicate or expand on our empirical findings.

Lastly, while our approach targets the Ethereum mainnet, differences in tooling, deployment practices, or proxy patterns across other EVM-compatible chains may limit its direct applicability. As such, we leave evaluation on other networks as future work.

[11] https://www.gnosis.io/

5.9.2 Internal Validity

To mitigate threats to internal validity, we explicitly accounted for code cloning on Ethereum when analyzing the prevalence of proxy contracts in RQ1 by introducing the notion of use cases (Section 5.5). This grouping helped to control for code cloning or templated contracts, ensuring a more accurate representation of unique proxy use cases. We carefully designed our sampling strategies to ensure representativeness. Specifically, we randomly selected contracts across a wide temporal range and usage scenarios to minimize bias in prevalence or classification findings. Additionally, our statistical procedures, including hyperparameter choices and hypothesis testing, were selected based on the characteristics of our data and standard best practices.

Additionally, to reduce the risk of missing relevant contracts in our DApp analysis (Section 5.8), we collected contract lists from three independent marketplaces. This triangulation aimed to improve coverage and minimize selection bias.

For the manual classification of proxy purposes in RQ3, we followed best practices from qualitative research, such as open coding, and employed the detection strategy proposed by Salehi et al. (2022). However, during our study, we identified several limitations in this prior approach. Notably, Salehi et al. (2022) did not empirically evaluate their method, and the method currently supports only three variants of Upgradeable Proxy Contracts (UPCs), missing other widely used variants. While this limitation highlights an opportunity for future work in improving detection tools, it may have affected our findings in Observation 15, especially regarding coverage of less common or newer UPC variants. We acknowledge this as

a potential threat to internal validity, suggesting the need for more comprehensive detection frameworks.

To ensure methodological rigor, we randomly sampled a representative use cases for our analysis. Before performing classification, we reviewed a broad range of proxy types described in the literature (Jorge Izquierdo 2018; Peter Murray 2018a; Santiago Palladino 2019b; OpenZeppelin 2018a; Gabriel Barros 2019b), which guided our coding schema. Lastly, our research prior expertise in smart contracts (Oliva et al. 2020; Kondo et al. 2020; Zarir et al. 2021; Oliva and Hassan 2021; Pacheco et al. 2023a; Pacheco et al. 2023b) helped ensure accurate and consistent classification judgments.

5.9.3 External Validity

The dataset used in this study includes *all* smart contracts and transactions available on Etherscan as of September 1st, 2022. As such, it provides a comprehensive snapshot of the Ethereum blockchain at that point in time. However, smart contract development practices evolve rapidly, and contracts created after this cutoff date may exhibit different characteristics. Therefore, our findings may not generalize to contracts deployed in the future.

Additionally, our study focuses exclusively on Ethereum. While Ethereum is the most widely used programmable blockchain, the patterns and prevalence of proxy contracts may differ significantly on other platforms (e.g., BNB Smart Chain, Polygon, Avalanche), limiting the generalizability of our results beyond Ethereum.

Because our detection method targets active proxy contracts, those whose proxy behavior is exercised through transactions, our findings may not fully apply to inactive proxy contracts. These contracts may exhibit different design patterns or use cases that are not captured by our analysis.

5.10 Conclusion

The proxy pattern is a foundational design technique in software engineering, valued for promoting modularity, reusability, and maintainability. Within the realm of programmable blockchains like Ethereum, it has become an important mechanism for enabling smart contract evolutions and minimizing deployment costs. Yet, this added flexibility comes with notable drawbacks. While proxy contracts facilitate contract evolution and extensible architectures, they can also obscure the actual execution logic, reduce code transparency, and complicate auditing undermining core principles of decentralization and trust. In this way, the proxy pattern functions as a double-edged sword: it empowers developers with upgradeability and cost-efficiency, but at the potential expense of clarity, security, and stakeholder confidence. Despite its widespread adoption, the nuanced trade-offs of proxy-based designs in blockchain ecosystems remain underexplored.

In this chapter, we proposed a scalable approach for detecting proxy contracts based on their fundamental conceptual properties rather than predefined patterns or heuristics. Unlike prior techniques that often rely on `fallback` function analysis, transaction replay, or manual inspection, our method leverages historical transaction traces to accurately identify proxy contracts at scale. Specifically, we focus on

two key characteristics of proxy contracts (Section 2.15.4) for detecting proxies contracts. Its key benefits include scalability, precision, and the ability to detect a wide variety of proxy contracts, including those that do not rely on `fallback` functions, thus offering broader and more reliable detection than existing approaches.

We then used our method to test the hypothesis that proxy contracts constitute a significant proportion of deployed Ethereum smart contracts. We conducted the first large-scale empirical study of the proxy pattern in Ethereum, analyzing a dataset of over 50 million smart contracts with 1.6B transactions. Our study aimed to quantify its prevalence (RQ1), examine the diversity of integration strategies (RQ2), and classify our detected proxy contracts based on their purpose and different implementation types (RQ3).

For RQ1, we found that over 14% of deployed contracts are active proxies, which confirms that proxy contracts are indeed a significant and substantial part of the Ethereum ecosystem. Our longitudinal analysis across activity level, use case, stakeholder adoption, and transaction flow reveals a clear upward trend in proxy usage, emphasizing their growing role in enabling modular and maintainable designs on Ethereum. For RQ2, we identified 12 distinct creational patterns for proxy deployment, categorized into on-chain and off-chain styles. On-chain deployments dominate, representing 99.3% of all proxy instances and 60.9% of use cases. Notably, non-singleton deployments are overwhelmingly on-chain (94.5%), which are also associated with greater diversity and scale. For RQ3, we first manually classified proxies into Forwarder Proxy Contracts (67.8%) and Upgradeable Proxy Contracts (32.2%). While 79% of use cases follow established standards, especially ERC-1167 minimal proxy contracts (29.4%) for cheap contract cloning and

cost reduction, over 21% adopt custom designs, highlighting significant diversity in real-world proxy adoption and the need for improved monitoring tooling and awareness.

Beyond our core findings, we examined the trade-offs of proxy contracts (Section 5.8.1.1) and the two deployment styles (Section 5.8.2). In addition, we outlined (i) the importance of adopting a system-level perspective when analyzing smart contracts, (ii) demonstrated how proxy contracts can obscure logic and hinder transparency in DApps, (iii) provided a structured decision flowchart to guide developers in choosing between on-chain and off-chain deployment styles, and (iv) outlined a research agenda with six open questions calling for better detection tool to detect the type of proxy contracts (Section 2.15.5), deeper understanding of upgrade patterns, and scalable governance models aligned with DApp complexity and user base.

In addition, of the 10,549 SATD contracts identified in Section 4.4, 60 (0.56%) are linked to 6,794 proxy contracts. This reuse is made possible by the proxy pattern's data separation principle, in which the proxy contract maintains the state while the associated logic contract (i.e., the SATD contract) remains stateless. As a result, the same logic contract can be shared across multiple proxy contracts, each operating as an independent instance with its own execution context and persistent storage (Section 2.15). This reveals a key issue.

Beyond the conventional propagation of SATD through traditional code cloning activities (Section 4.5), SATD can also propagate through a different channel, runtime logic cloning, in which multiple proxy contracts delegate their execution to a

shared logic contract which contain SATD. When a logic contract suffers from internal quality issues (e.g., bugs, technical debt, or vulnerabilities) these issues are propagated to every proxy contract that shares the same logic. As such, a single flaw in the shared logic can contaminate thousands of proxies simultaneously. This not only amplifies the potential impact of SATD but also complicates its resolution. While remediation is only feasible if the proxy contracts are of the upgradeable proxy contracts (UPCs) type, even then, maintaining SATD-bearing logic remains highly challenging due to the need for coordinated upgrades across many proxies, each potentially governed by different stakeholders, access controls, or upgrade policies. Recognizing this propagation channel is essential for understanding the broader implications of SATD and its diffusion through proxy contracts within the Ethereum ecosystems. We leave this as a promising research avenue for future studies.

Furthermore, our findings reveal that just under one-third of all use cases, and 86% of the top 50 most active DApps, employ proxy contracts to enable contract evolution. However, upgradeable proxy contracts (UPCs) introduce a range of critical risks, including malicious upgrades, centralization of control, governance vulnerabilities, and private key compromises. They also increase the potential for storage incompatibility across upgrades, obscure transparency for end users, and complicate security audits due to the separation between proxy and logic contracts. These risks highlight the urgent need for more robust, transparent, and versatile detection mechanisms that are applicable to all contracts and capable of identifying diverse variants of UPCs (i.e., UPC patterns). This need is further underscored by the limitations of existing methods, which often lack both breadth and depth of

detection (Observation 4). Moreover, accurate detection of UPCs is essential to facilitate a deeper understanding of how smart contracts evolve in practice, whether to address bugs, resolve technical debt as mentioned earlier, or mitigate vulnerabilities, thereby supporting both research and tool development in Ethereum.

Given the importance of such detection methods, and to support future studies to investigate the aforementioned challenges more accurately and comprehensively, the next chapter introduces UPC Sentinel, a precise approach for identifying the type of a proxy contract (whether it is a UPC or FPC), as well as the specific UPC pattern it conforms to, all without requiring access to source code.

CHAPTER 6

UPC Sentinel: An Accurate Approach for Detecting Upgradeable Proxy Contracts in Ethereum

6.1 Introduction

Since its inception, Ethereum has revolutionized the blockchain landscape by introducing smart contract technology. A smart contract is self-executing software with the terms of the agreement directly written into lines of code (Tikhomirov et al. 2018; Oliva and Hassan 2021; Chen et al. 2020; Ebrahimi et al. 2023). Beyond mere value transfer, the primary function of traditional blockchain platforms like Bitcoin, Ethereum's smart contracts enable the development of full-fledged decentralized applications (DApps). DApps are significant because they offer a level of

autonomy, security, and transparency that traditional centralized applications cannot (Section 2.4). By operating on a blockchain platform, they minimize the risks of data breaches and downtime, resist censorship, and give users enhanced control over their data and transactions, fostering trust and collaboration in digital interactions. Today, DApps support various business domains, including decentralized finance (DeFi) (Loo 2025; Fortune Business Insights 2023) and supply chain management (Deloitte 2023).

In blockchain platforms like Ethereum, the principle of immutability ensures transparency, trust, and security by guaranteeing that transactions cannot be rolled back and the bytecode of a deployed contract cannot be changed (Section 2.5). However, immutability comes at a significant cost. As Decentralized Applications (DApps) grow more complex, the need for periodic maintenance, such as bug fixes and feature enhancements, becomes increasingly important (Chen et al. 2021b; Chen et al. 2020). In a rapidly evolving technological landscape, where deficiencies (e.g., vulnerabilities, technical debt, or bugs) may be discovered post-deployment, immutability can hinder timely updates, potentially compromising functionality or security. As a result, developers face a delicate trade-off between the assurance provided by immutability and the practical need for application maintenance and evolution.

Despite being the most prominent infrastructure for hosting and executing smart contracts, Ethereum does *not* provide any native mechanism to address the need for DApp maintenance and evolution. To bridge this gap, practitioners have designed and implemented various mechanisms for evolving (i.e., upgrading) smart contracts, among which the *proxy pattern* method stands out due to its flexibility in

enabling the maintenance of DApps (Ethereum 2023a; Salehi et al. 2022; Ebrahimi et al. 2023). Inspired by the proxy design pattern (Gamma et al. 1994), a proxy smart contract enables contract evolution by separating the contract logic from its state. A *proxy contract* delegates calls to a *logic contract* (also referred to as the *implementation contract*) that contains the actual logic code, while the proxy itself maintains the state. If the logic needs to be updated, (i) a new logic contract is deployed, and (ii) the proxy is set to redirect calls to this new contract instead of the old one, allowing the original logic contract to be upgraded without losing its state or changing its blockchain address (Section 2.15.5). Note that we may use the terms "logic contract" and "implementation contract" interchangeably throughout the remainder of this chapter.

Given the increasing popularity of the proxy pattern in Ethereum smart contracts (Section 5.5) and the associated risks it introduces (Section 5.1), there is a pressing need for tools that can accurately distinguish between Upgradeable Proxy Contracts (UPCs) and Forwarder Proxy Contracts (FPCs) (Santiago Palladino 2019b).

In the previous chapter, we introduced a method capable of identifying whether a given contract functions as a proxy. While effective for general proxy detection, a more fine-grained approach is required to accurately classify the proxy type, specifically, whether it is a UPC or an FPC (Ebrahimi et al. 2023), and, in the case of UPCs, to further identify the specific UPC variant in use. As DApps continue to grow in complexity and scale, ensuring the transparency, security, and reliability of their underlying proxy infrastructure becomes increasingly critical. For end-users and stakeholders, such tools enhance transparency by revealing the presence of proxy-based contract evolution mechanisms that may alter contract behavior over time.

For regulators and auditors, they enable compliance monitoring by helping detect whether the evolution mechanism is properly implemented, free of bugs, or potentially misused. Finally, for researchers, these tools are essential for understanding how real-world DApps are maintained post-deployment, and how their evolution practices compare to those of traditional centralized software systems (Santiago Palladino 2019b; Ebrahimi et al. 2023; Salehi et al. 2022; Bodell III et al. 2023).

Several studies have proposed methods for detecting UPCs; however, these methods often fall short either in terms of detection depth, such as the number of UPC patterns they can identify, or breadth, the range of contracts to which they can be applied. The majority of these approaches rely on source code analysis (Bodell III et al. 2023; Qasse et al. 2023; Feist et al. 2019). However, the applicability of source-level UPC detectors is inherently limited, as most smart contracts on Ethereum are closed-source (Xu et al. 2020; Huang et al. 2021; Li et al. 2024a; Chen et al. 2019; Chen et al. 2021c; Chen et al. 2022b). In contrast, bytecode-based methods are not constrained by source code availability. Yet, to the best of our knowledge, only one study has specifically tackled the detection of UPCs at the bytecode level (Salehi et al. 2022). However, while prior studies enumerated several UPC patterns (Meisami and III 2023), existing bytecode-level detection efforts, such as that of Salehi et al. (2022), are limited to just three specific UPC patterns: Regular Upgradeable Proxies (Palladino 2021), Beacon Beacon proxies (Dharma 2019), and the Universal Upgradeable Proxy Standard (UUPS) (Gabriel Barros 2019b). In addition, their work lacks empirical evaluation at scale, leaving open questions about the generalizability and practical effectiveness of their method compared to source-level counterparts.

From a methodological perspective, existing methods suffer from limited integration of static and dynamic analysis techniques, restricting their ability to capture complex UPC patterns that span multiple contract layers. Static analysis alone often lacks sufficient context, particularly when roles and responsibilities are distributed across independently deployed contracts. In such cases, only dynamic analysis can reveal the necessary dependencies, effectively expanding the static analyzer's search space. Conversely, dynamic-only approaches can also fall short, as indications of the evolution mechanism may not manifest at the behavioral level, for example, if no upgrade has occurred, resulting in the absence of relevant dynamic traces. Effective detection of UPCs therefore requires proper orchestration between static and dynamic techniques. While a few existing methods attempt to combine the two, current approaches face coordination challenges. These include analyzing only one dependency (e.g., the latest implementation contract) at a time, covering only a limited set of patterns, or focusing solely on immediate dependencies. As a result, they often fail to detect UPCs embedded in transitive dependencies (i.e., dependencies of dependencies). Such deeper, multi-level dynamic dependency analysis remains largely overlooked in the literature.

To address the limitations of current methods, we propose UPC Sentinel, a three-layer algorithm that combines static and dynamic analysis techniques on smart contract bytecode to detect and classify proxy contracts. UPC Sentinel is informed by a comprehensive survey of widely adopted Upgradeable Proxy Contract Patterns (UPC Pattern) (Meisami and III 2023). Given an input contract, the algorithm first performs dynamic analysis (Section 5.4) to determine whether the contract is an active proxy, that is, a proxy whose functionality has been exercised at least once

since its deployment (Ebrahimi et al. 2023). If the contract is identified as an active proxy, UPC Sentinel applies a coordinated static and dynamic analysis to classify the proxy as either a UPC or an FPC. If the proxy is classified as a UPC, the final stage performs a fine-grained classification to identify the specific UPC Pattern the UPC conforms to.

We carefully and empirically evaluated UPC Sentinel using two independent ground truth datasets, namely GE and GB, to assess its performance from multiple perspectives. We also compared its effectiveness against the state-of-the-art method, USCHUNT (Bodell III et al. 2023), which operates at the source code level. Specifically, we evaluated UPC Sentinel through the following four research questions:

RQ1: How accurately does UPC Sentinel detect Upgradeable Proxy Contracts in the GE dataset? To establish a baseline for evaluating real-world applicability, we assess UPC Sentinel's accuracy in detecting Upgradeable Proxy Contracts using GE, a diverse, high-quality dataset of confirmed UPCs.

RQ2: How does UPC Sentinel's proxy detector compare to USCHUNT in detecting proxy contracts? Accurate proxy detection is essential for identifying upgradeable contracts. This question benchmarks UPC Sentinel's proxy detection component against the USCHUNT method to evaluate its reliability and effectiveness in identifying proxy contracts.

RQ3: How does UPC Sentinel's proxy type classifier compare to USCHUNT in classifying proxy contracts? Beyond detecting proxies, classifying their type, whether upgradeable or forwarder, is critical for understanding their role in contract evolution. This question evaluates the accuracy and applicability of UPC Sentinel's proxy type classifier in comparison to USCHUNT.

RQ4: How does UPC Sentinel's UPC pattern classifier compare to USCHUNT in classifying UPC patterns? Several known pattern exists for detecting implementing upgradeable proxy contracts. Identifying the specific UPC pattern employed by a proxy provides deeper insight into its design and potential security implications. This question assesses UPC Sentinel's ability to perform fine-grained UPC pattern classification and contrasts its performance with that of USCHUNT.

Contributions. (i) We introduce UPC Sentinel, an effective and accurate algorithm for identifying and classifying proxy contracts at the bytecode level. In particular, it determines whether a proxy is an Upgradeable Proxy Contract (UPC) or a Forwarder Proxy Contract (FPC). **(ii)** UPC Sentinel is centered on three derived bytecode-level reference architectures for upgradeable proxy contacts, grounded in the taxonomy of upgradeable smart contract patterns curated by Meisami and III (2023). **(iii)** Our method enables fine-grained identification of the specific UPC Pattern to which a detected UPC conforms. **(iv)** We compile a new ground truth dataset containing over 3,177 verified UPCs. **(v)** We identify and correct labeling inconsistencies in a dataset released by Bodell III et al. (2023), which includes over 994 contracts. **(vi)** We conduct a comprehensive empirical evaluation using both ground truths, covering more than 4,000 proxy contract data points.

Chapter structure: Section 6.2 outlines the design of UPC Sentinel. Section 6.3 describes our data sources, the characteristics of the ground truth datasets used, and the data quality checks we performed. Sections 6.4, 6.5, 6.6, and 6.7 present our empirical evaluations of UPC Sentinel. Section 6.8 discusses the practical value of UPC Sentinel and examines the associated time cost. Section 6.9 identifies potential

threats to the validity of our study and outlines future directions to address the current limitations. Finally, Section 6.10 concludes the chapter.

6.2 UPC Sentinel

6.2.1 Technical Requirements

The primary objective of UPC Sentinel is to accurately distinguish Forwarder Proxy Contracts (FPCs) from Upgradeable Proxy Contracts (UPCs) on the Ethereum blockchain. UPCs represent the most common mechanism for enabling smart contract evolution on Ethereum (Salehi et al. 2022). Accordingly, the detection of other evolution mechanisms, such as `selfdestruct`, the `CREATE2` metamorphic mechanisms, the strategy pattern, and others (Section 2.14), falls outside the scope of this thesis.

UPC Sentinel takes as input a smart contract address or a list of addresses. It first determines whether each input contract functions as a proxy, with an emphasis on achieving high recall and precision (**Requirement R1**). This ensures both comprehensive and accurate identification of proxy contracts. If a contract is confirmed to be a proxy, UPC Sentinel then classifies it as either an Upgradeable Proxy Contract (UPC) or a Forwarder Proxy Contract (FPC), based on whether it is specifically designed to support contract evolution (**Requirement R2**).

Furthermore, the majority of deployed smart contracts lack publicly available source code (Xu et al. 2020; Huang et al. 2021; Li et al. 2024a; Chen et al. 2019; Chen et al. 2021c; Chen et al. 2022b), limiting the applicability of UPC detectors that rely on source-level artifacts. Therefore, UPC Sentinel is designed to operate

directly on smart contract bytecode, enabling effective detection of UPCs without requiring access to the source code (**Requirement R3**).

Moreover, several Upgradeable Proxy Contracts Patterns (UPC Pattern), such as Transparent UPC Pattern, Beacon UPC Pattern, and ERC-1822 Universal UPC Pattern, have emerged as common patterns for enabling contract evolution (Section 2.15.6). While these UPC Patterns are all rooted in the same foundational proxy design, they differ in critical implementation aspects, such as how and where the implementation contract address is stored, or how upgrade functions are defined and invoked. Therefore, UPC Sentinel must not only maintain high detection accuracy across different patterns but also correctly identify the specific pattern to which each detected UPC conforms (**Requirement R4**).

6.2.2 Three-layered Design of UPC Sentinel

We now introduce the UPC Sentinel algorithm, designed to distinguish Upgradeable Proxy Contracts (UPCs) from Forwarder Proxy Contracts (FPCs) through a three-layered architecture. The input to UPC Sentinel is a list of smart contract addresses. Figure 6.1 illustrates the architecture, which comprises three distinct layers: **(i)** the Proxy Contract Detector Layer, **(ii)** the Proxy Contract Type Classifier Layer, and **(iii)** the Upgradeable Proxy Contract Pattern Classifier Layer.

The first layer, the Proxy Contract Detector Layer, determines whether a given contract functions as a proxy contract, fulfilling **Requirement R1**. The second layer, the Proxy Contract Type Classifier Layer, evaluates whether the identified proxy is designed for contract evolution (i.e., a UPC) or serves other purposes (i.e., an FPC), fulfilling **Requirement R2**. The third and final layer, the upgradeable proxy contract

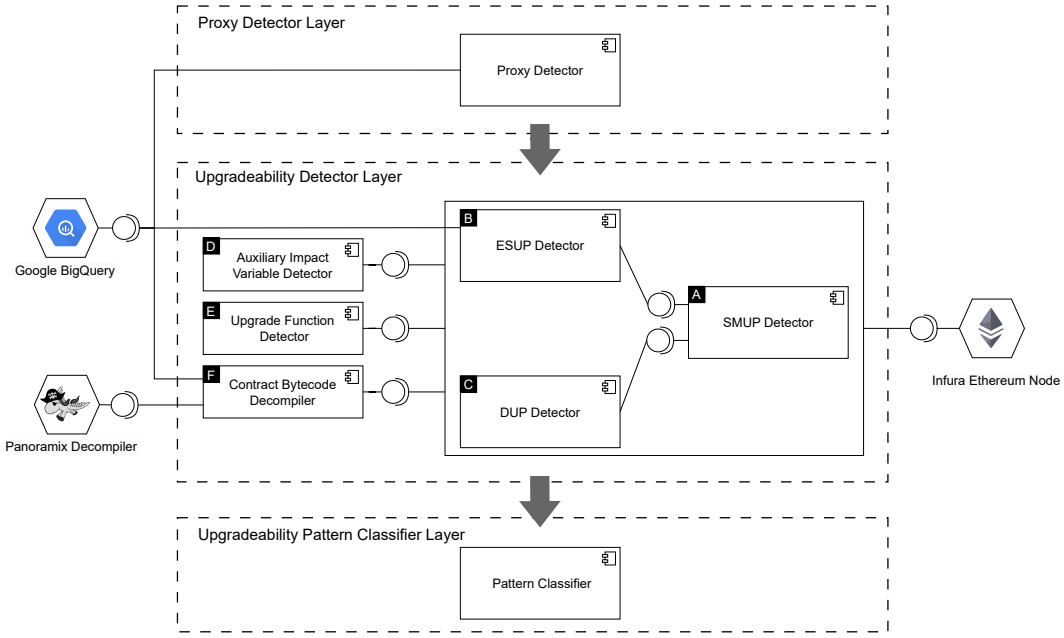


Figure 6.1: UPC Sentinel architecture.

pattern classifier, applies only if the proxy is identified as a UPC and performs a fine-grained classification to determine which specific pattern the UPC conforms to, fulfilling **Requirement R4**.

For each input contract address, UPC Sentinel outputs: (i) a binary label indicating whether the contract is a proxy, (ii) if so, whether it is a UPC or an FPC, and (iii) if it is a UPC, the specific UPC Pattern it implements. The following Sections 6.2.3, 6.2.4, and 6.2.5 provide a detailed discussion of each layer.

6.2.3 Proxy Contract Detector Layer

We employed the dynamic method introduced in Chapter 5 to detect proxy contracts Ebrahimi et al. (2023). Specifically, we identified two *inherent and conceptual properties* of proxy contracts, grounded in existing guidelines and standards (Gamma

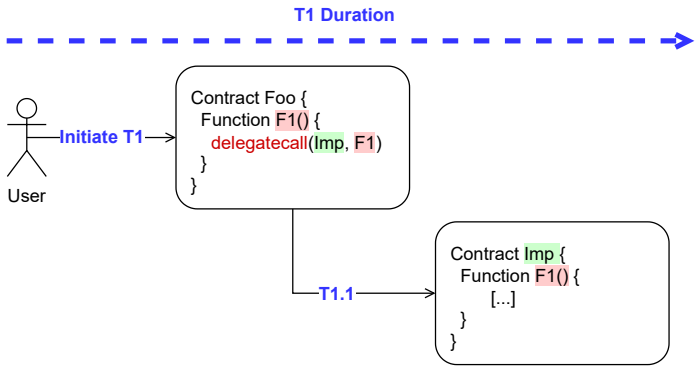


Figure 6.2: Illustrative example of our proxy detector leveraging its two inherent properties.

et al. 1994; Agarwal 2023; Wasnik 2024; Mudge 2020a; Fravoll 2018; OpenZeppelin 2017), and used them as filtering criteria. Our method analyzes the historical transactions of smart contracts and compares them against these criteria to identify proxy behavior. To illustrate how our proxy detector operates at the transaction level, consider the example in Figure 6.2. Suppose a user initiates a transaction session T1 that calls the function `F1` of the `Foo` contract. If, during T1, the `F1` function of `Foo` invokes a `delegatecall` (represented as internal transaction T1.1) to the `F1` function of the `Imp` contract, where both functions share the same signature, then `Foo` is identified as a proxy contract. For each input contract address, this layer outputs a binary label indicating whether the contract is a proxy and, if so, returns the list of implementation contracts it has ever delegated calls to.

There are two primary reasons for our choice of this method. First, this method is *not* based on source code analysis, which satisfies our R3 requirement. Second, we demonstrated that this method achieves perfect precision and recall in detecting `active` proxy contracts and detects more proxy contracts than a previous approach

by Salehi et al. (2022) (Section 5.4). We defined an active proxy contract as one that has had at least one transaction following its creation, in which the proxy delegates calls to its implementation contract (Section 5.9). Hence, the adoption of this method introduces a specific focus to our methodology, narrowing our scope to detecting only active UPCs (Section 6.9 for a discussion about the potential impacts of this design choice).

6.2.4 Proxy Contract Type Classifier Layer

After identifying a proxy, the next critical step is to determine whether it is an Upgradeable Proxy Contract (UPC) or a Forwarder Proxy Contract (FPC) (Section 2.15.5). To this end, we propose an algorithm that integrates both static and dynamic analyses to accurately classify the type of proxy contract. Specifically, we apply static analysis to the decompiled bytecode of the contract, thereby satisfying Requirement R3. In the following sections, we first describe how we derived three Upgradeable Proxy Contract Reference Designs (UPC Reference Designs) that guided the development of our bytecode-level classification algorithms. Next, we introduce each UPC Reference Design along with its corresponding detection strategy. Finally, we present the implementation details of UPC Sentinel's six components of the Proxy Contract Type Classifier Layer.

6.2.4.1 Deriving UPC Reference Designs

Given our objective of detecting UPCs at the bytecode level, we set out to analyze the underlying bytecode-level mechanisms used by established UPC Patterns and to abstract these mechanisms into higher-level representations, namely, Upgradeable

Proxy Contract Reference Designs (UPC Reference Designs). By focusing on the identification of these abstractions, our goal is to detect any UPCs that conform to them, regardless of their specific implementation details (e.g., storage layout, inheritance hierarchy, variable types, etc.) or the availability of source code.

The input to this study consists of eight established standards and widely recognized UPC Patterns that are commonly endorsed by the Ethereum community. In particular, Meisami and III (2023) compiled a curated list of proxy patterns, which served as our starting point. From this list, we selected the eight patterns specifically designed for upgradeable proxy contracts. A comprehensive technical discussion of these patterns is provided in Section 2.15.6.

To identify a given contract as UPC, it must satisfy two key conditions at an abstract level (Johnson 2016; AL Amri et al. 2023; OpenZeppelin 2018b; OpenZeppelin 2018f; Ethereum 2023b): i) the contract must function as a proxy, and ii) it must include an upgrade mechanism (typically a function) that allows the modification of the implementation (logic) contract address stored within the proxy. While the first condition is addressed by the Proxy Contract Detector Layer (Section 6.2.3), the second condition encapsulates the core essence of upgradeability. Consequently, identifying the upgradeability status of a proxy contract (i.e., whether a given proxy is a UPC or an FPC) can be formulated as a search for two critical syntactic features: (i) the location where the state variable storing the proxy's implementation contract address is declared, and (ii) the location where the upgrade function enabling the modification of this state variable is defined. We briefly call the former and latter features as the "implementation variable location" and the "upgrade function location", respectively.

Given the critical role of these two features in determining the upgradeability status of a proxy contract, we aim to analyze how they manifest across various UPC Patterns and to derive a generalized, guided approach for detecting the type of a proxy contract at the bytecode level. Notably, Bodell III et al. (2023) conducted an extensive study on a wide range of syntactic features associated with UPC Patterns at the source code level, including the two features mentioned above. While their findings are foundational and inform our analysis, the transformations that occur during compilation, such as inheritance linearization, storage layout standardization, and the unification of read/write operations, necessitate a deeper investigation into how these features are encoded and potentially altered at the bytecode level.

To analyze each UPC Pattern, the first and third authors independently reviewed its online documentation and performed deductive coding based on the two key features and the initial (source code-level) values reported in Bodell III et al. (2023)'s study. They then met to reconcile their results and found no disagreements in their classifications. Table 6.1 summarizes the values, similarities, and differences of these two features across all eight patterns, comparing our bytecode-level analysis with the source code-level findings of Bodell III et al. (2023). Notably, in three patterns, namely, the Inherited Storage UPC Pattern, the Eternal Storage UPC Pattern, and the ERC-2535 Diamonds UPC Pattern, we observed discrepancies between bytecode-level and source code-level values. However, for the remaining five patterns, our findings align with those of the prior study (Bodell III et al. 2023).

In particular, for the first feature, namely, the implementation variable location, the documentation of both the Inherited Storage UPC Pattern (OpenZeppelin 2018d) and the Eternal Storage UPC Pattern (OpenZeppelin 2018c) indicates

Table 6.1: Comparison of implementation variable location and upgrade function location across UPC Patterns, analyzed at the source code (top) and bytecode (bottom) levels. Red-highlighted values indicate discrepancies between the two analysis levels (Bodell III et al. 2023).

Study	Feature	UPC Patterns							
		Inherited Storage Upgradeable Proxy	Eternal Storage Upgradeable Proxy	ERC-1967 Upgradeable Proxy (Unstructured Storage)	Transparent Upgradeable Proxy	ERC-2535 Diamonds Upgradeable Proxy	ERC-1822 Universal Upgradeable Proxy	Beacon Upgradeable Proxy	Registry Upgradeable Proxy
Bodell III et al. (2023) (Source code level)	Implementation variable location	Inherited	-	Proxy	Proxy	Proxy	Proxy & Implementation	External	External
	Upgrade function location	-	-	Proxy	Proxy	Implementation	Implementation	External	External
Our Study (Bytecode level)	Implementation variable location	Proxy	Proxy	Proxy	Proxy	Proxy & Implementation	Proxy & Implementation	External	External
	Upgrade function location	Proxy	Proxy	Proxy	Proxy	Implementation	Implementation	External	External

that the proxy contracts inherit from another contract in which the upgradeability-related state variables are declared. At the source code level, this implies that the implementation variable resides in the inherited contract for both patterns. Bodell III et al. (2023) also reported this for the former pattern. However, when examining the compiled bytecode, Solidity applies the C3 linearization method to flatten the inheritance hierarchy into a linear order (Wikipedia.org 2014). This process merges the storage layouts of parent contracts into the proxy's bytecode. Consequently, for both the Inherited Storage and Eternal Storage UPC Patterns, the implementation variable is ultimately embedded within the proxy's bytecode.

In the case of the ERC-2535 Diamonds UPC Pattern (Mudge 2020c; Mudge 2020a), the Diamond Proxy maintains a mapping (i.e., the implementation variable) of function selectors to their corresponding implementation (also known as facet) contract addresses. The upgrade process is orchestrated by a key implementation contract called the DiamondCutFacet. To ensure that the upgrade function can effectively modify the Diamond Proxy's mapping state variable, both the Diamond Proxy and the DiamondCutFacet must adhere to a consistent storage layout. This consistency is achieved by defining the mapping state variable in a shared library used by both the Diamond Proxy and the DiamondCutFacet. As a result, both the proxy and its implementation contract responsible for managing the upgrade mechanism have access to the implementation variable at the same storage slot. The upgrade must be initiated by the Diamond Proxy via a `delegatecall`, ensuring that changes to the mapping occur within the proxy's storage context. At the bytecode level, since Solidity compiles the shared library code into both the proxy and the implementation contracts, the implementation variable is present in the bytecode of both the Diamond Proxy and the DiamondCutFacet.

Following the deductive coding process (Hsieh and Shannon 2005), we further abstracted the UPC Patterns into three distinct groups, referred to as Upgradeable Proxy Contract Reference Designs (UPC Reference Designs), based on the bytecode-level values of the two key features. This abstraction provides a higher-level perspective that transcends the specific source code implementation details of individual patterns. By focusing on core characteristics and bytecode-level representations, these reference designs enable a robust and generalizable detection method capable of identifying UPCs whose upgrade mechanisms align with one of the three

designs, regardless of implementation specifics. This approach offers a resilient methodology that reduces the need for case-by-case reanalysis and is more adaptable to minor variations that may emerge over time.

The first UPC Reference Design, Self-Managed Upgradeable Proxy (SMUP), includes the Inherited Storage UPC Pattern, Eternal Storage UPC Pattern, ERC-1967 UPC Pattern (Unstructured Storage), and Transparent UPC Pattern. In these designs, both the implementation variable and the upgrade function are embedded within the proxy's bytecode. The second UPC Reference Design, Delegated Upgradeable Proxy (DUP), consists of the ERC-1822 Universal UPC Pattern and ERC-2535 Diamonds UPC Pattern. In this group, the implementation variable is present in both the proxy and the implementation bytecode, while the upgrade function resides in the implementation bytecode. The final UPC Reference Design, Externally Supervised Upgradeable Proxy (ESUP), encompasses the Beacon UPC Pattern and Registry UPC Pattern, in which both the implementation variable and the upgrade function are managed externally, residing in a separate contract's bytecode. Notably, this final reference design was also identified in Bodell III et al. (2023)'s study, where it is referred to as the Registry Patterns. In the following sections, we provide a detailed explanation of each reference design.

Figure 6.3 depicts the structure of the three UPC Reference Designs and their respective variants. In the following, we introduce each UPC Reference Design in detail.

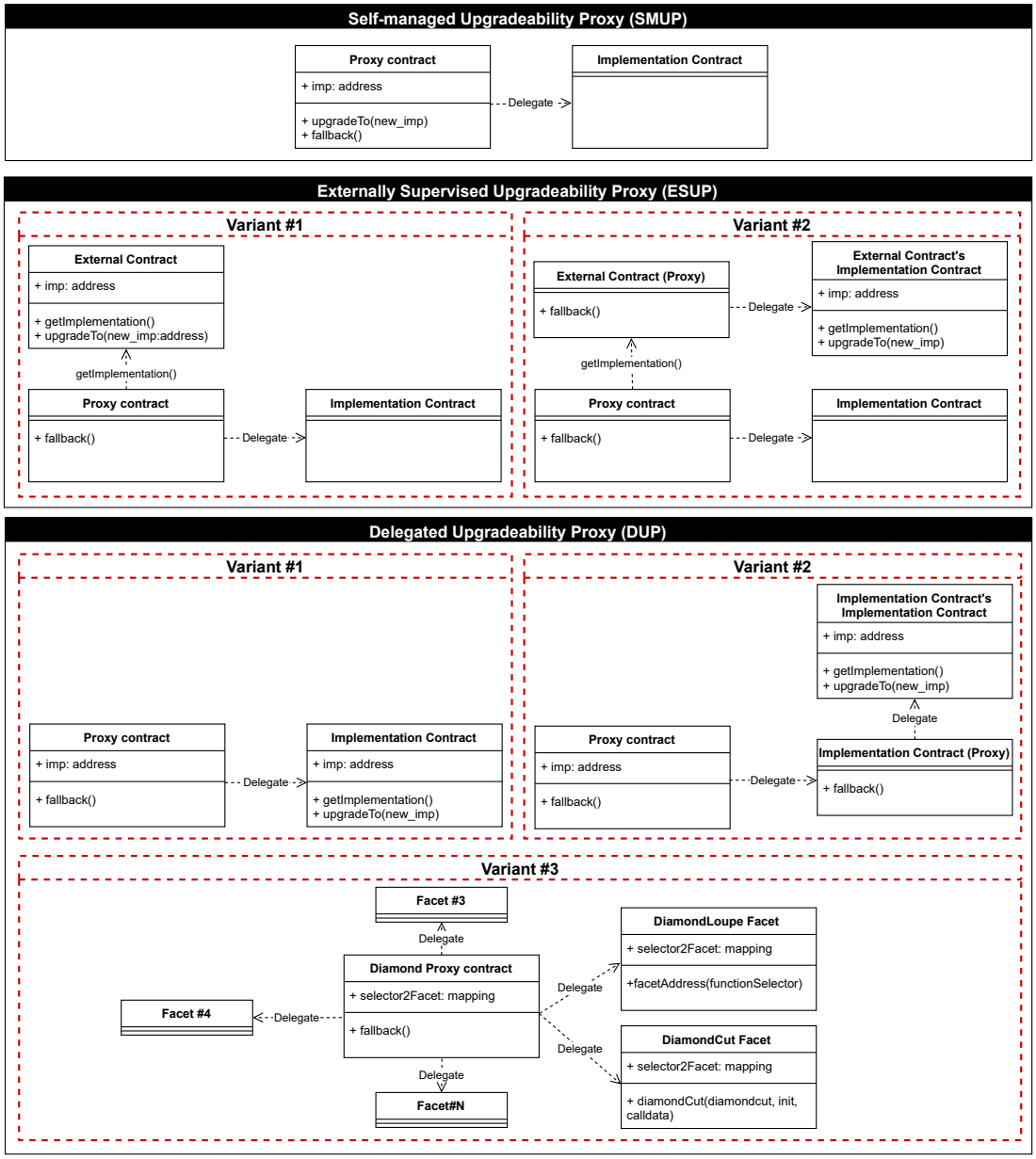


Figure 6.3: Structure of the three Upgradeable Proxy Contract Reference Designs (UPC Reference Designs) and their respective variants. Note that all identifiers shown (e.g., function names and variable names) are illustrative; arbitrary identifiers may be used in practice.

6.2.4.1.1 SMUP. The first and most common reference design is Self-Managed Upgradeable Proxy (SMUP), which occurs when the proxy contract itself implements the upgrade function. Figure 6.3 illustrates a simplified UML class diagram of this UPC Reference Design. In this case, the proxy contract defines at least one external or public function that allows the existing implementation contract address, stored in the proxy's storage, to be replaced with a new one. Examples of this class include the Transparent UPC Pattern, Inherited Storage UPC Pattern, Eternal Storage UPC Pattern, and ERC-1967 UPC Pattern (Unstructured Storage). This type can be detected by applying static analysis directly to the proxy contract's bytecode.

6.2.4.1.2 ESUP. The second reference design is the Externally Supervised Upgradeable Proxy (ESUP), which occurs when both the implementation variable and the upgrade function are located in an external contract upon which the proxy contract depends. Figure 6.3 illustrates a simplified UML class diagram of this UPC Reference Design. In this setup, the proxy typically queries the external contract to retrieve the address of the latest implementation contract before delegating calls to it. The Beacon Upgradeable Proxy and Registry Upgradeable Proxy are two common examples of this design. Detecting this type requires a dynamic method to first identify the external contract address. Once the external contract is located, a static analyzer can examine its bytecode to determine whether it implements the upgrade function (variant #1). In more complex scenarios, the external contract itself may be a proxy, with the upgrade function implemented in its own implementation contracts (variant #2).

6.2.4.1.3 DUP. The third reference design is Delegated Upgradeable Proxy (DUP), which occurs when the upgrade function is implemented within the proxy's implementation contracts. Figure 6.3 illustrates a UML class diagram of this UPC Reference Design. In this design, the proxy stores the address of its implementation contract in a specific storage slot, referred to as the `implementation slot`. The implementation contract also defines an identical `implementation slot` in its own storage layout. During the upgrade process, the proxy delegates the upgrade call to its implementation contract, which contains the upgrade function. Since the call is executed in the proxy's context via `delegatecall`, any modification to the implementation slot in the upgrade function updates the proxy's storage accordingly (Section 2.15.6 for an example). ERC-1822 Universal UPC Pattern and ERC-2535 Diamonds UPC Pattern are representative patterns of this class.

To detect this type, it is essential to first identify all of the proxy's implementation contracts. Each of these implementations must then be statically analyzed to determine: (i) whether it defines a matching implementation slot and (ii) whether it includes an upgrade function (variant #1). As with ESUP, a more complex case arises if the implementation contract is itself a proxy. In this variant (#2), the detection procedure must be recursively applied to the proxy's implementation's own implementation contracts (i.e., secondary dependencies).

Regarding the ERC-2535 Diamonds UPC Pattern, the Diamond proxy maintains a mapping between function selectors and their corresponding facet contract addresses. According to the standard (Mudge 2020c; Mudge 2020a), the `DiamondCutFacet` is the facet responsible for executing the upgrade function, typically named `diamondCut`. Since the Diamond proxy interacts with its facets via

`delegatecall`, any invocation of the `diamondCut` function must be delegated to the `DiamondCutFacet`. However, if the Diamond proxy has never initiated a transaction involving the `DiamondCutFacet`, it becomes infeasible to detect this facet using conventional transaction analysis alone.

To overcome this limitation, we propose an alternative dynamic method based on another standard facet in the pattern, the `DiamondLoupeFacet`. This facet functions as an introspection tool, enabling queries about the Diamond's registered facets and their functions. Specifically, when the `facetAddress` function is called on the `DiamondLoupeFacet` with a given function selector as input, it returns the facet contract address where that selector is implemented. Since the function selector for `diamondCut` is known a priori, we can query the `DiamondLoupeFacet` to retrieve the facet address associated with it. If the returned address corresponds to the `DiamondCutFacet`, then the presence of the upgrade function is confirmed, and the proxy is deemed upgradeable.

We classify this detection strategy as the third variant of the Delegated Upgradeable Proxy (DUP) reference design (variant #3), as it introduces an additional behavioral analysis mechanism that supplements the traditional bytecode and transaction-based approaches.

6.2.4.2 Core Components

This section provides a detailed explanation of the six components that constitute the Proxy Contract Type Classifier Layer, as illustrated in Figure 6.1.

6.2.4.2.1 SMUP Detector. This component analyzes the proxy contract to identify signs of upgradeability. Figure 6.4 illustrates our algorithm. The component

takes as input a proxy contract along with all its corresponding implementation contracts as detected by the Proxy Contract Detector Layer. First, we examine whether any of the implementation contract addresses are hard-coded within the proxy's bytecode. If an implementation address is hard-coded, it indicates that the proxy is not upgradeable. For each implementation contract that is not hard-coded, we record its address for further analysis (A1). If all implementation addresses are hard-coded, the proxy is labeled as a Non-UPC (i.e., a FPC proxy). Otherwise, we continue to the next step (A2).

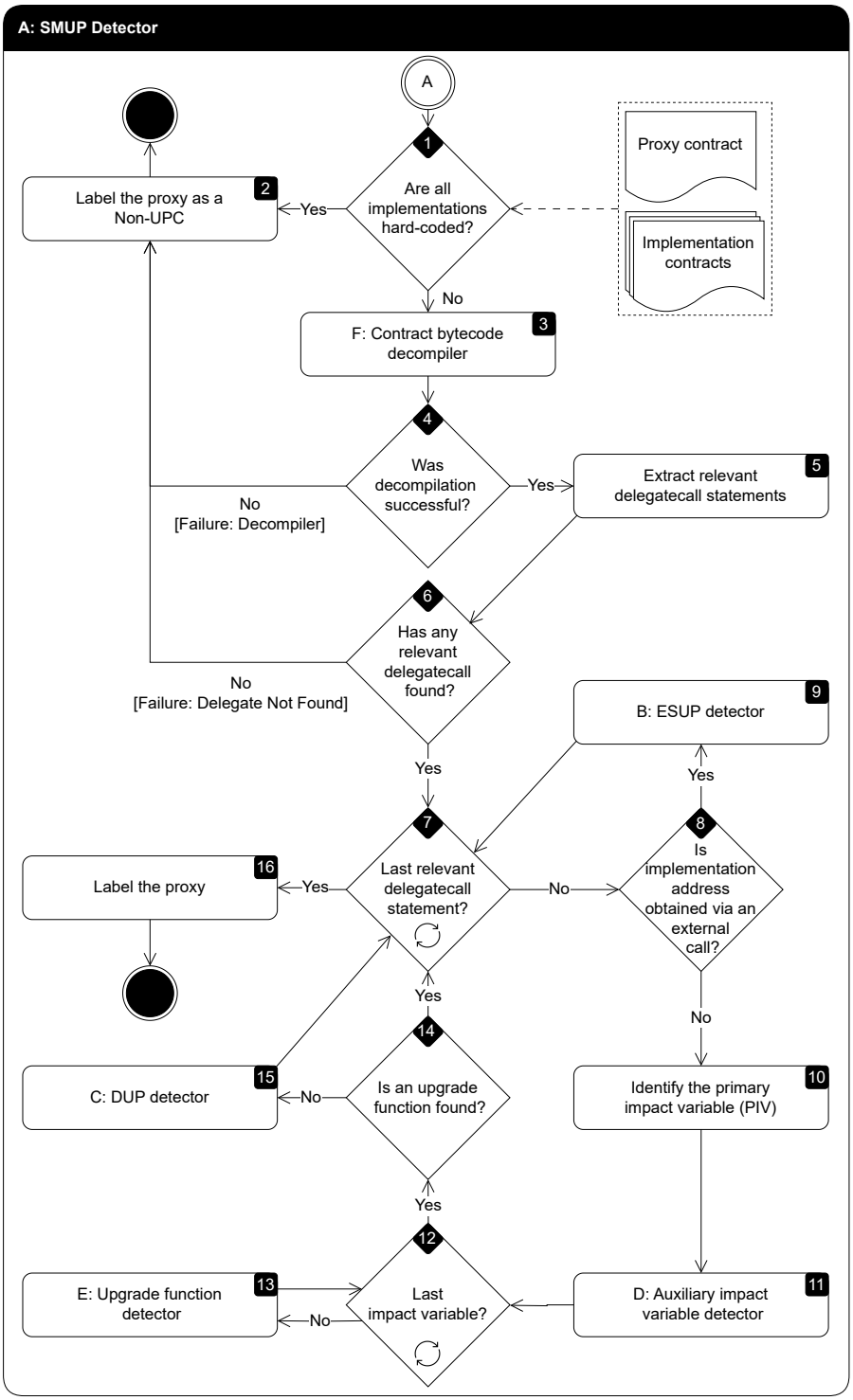


Figure 6.4: Detailed design of the “SMUP Detector” component.

Next, the proxy contract address is passed to the "Contract Bytecode Decompiler" component, which retrieves the proxy's bytecode and attempts to decompile it using the Panoramix decompiler (Palkeo 2019) (A3). If Panoramix fails to generate a decompiled source file, the process is halted with the "Failure: Decompiler" flag, and the proxy is labeled as Non-UPC. If successful, the analysis proceeds to the next step (A4). In the next step (A5), we identify where proxy functionality is implemented at the decompiled bytecode level by extracting all `relevant delegatecall` statements. As discussed in Section 6.2.3, proxy contracts exhibit two fundamental properties, Property #1 and Property #2. Since bytecode is a machine-level representation of source code, these properties should be observable in the decompiled bytecode of any valid proxy contract. Accordingly, every proxy contract is expected to contain at least one function that uses a `delegatecall` to process the `calldata` at the bytecode level.

To validate this expectation, we conducted an automated analysis of the decompiled bytecode from a statistically significant random sample of 16,603 active proxy contracts, drawn as of September 2022 (at a 99% confidence level with a 1% margin of error). We identified bytecode features corresponding to Property #1 and Property #2 in 16,600 cases (99.98%), with only 3 exceptions (0.02%) representing rare edge cases. Figure 6.5 illustrates the bytecode features we identified for Property #1 and Property #2. For Property #1 (highlighted in green), a proxy function must include a `delegatecall` statement within a function. For Property #2 (highlighted in yellow), the selector of the implementation contract's function to which calls are delegated must either:

- be `call.data[0 len 4]` or `call.data[return_data.size len 4]`, indicating the use of the first four bytes of the `calldata` to determine the function to which calls must be delegated.
- alternatively, if both the proxy and implementation function selectors are identical and hard-coded, this suggests a specialized proxy function beyond the typical `fallback` implementation.

```
# Pattern #1
def _fallback() payable:
  delegate target with:
    funct call.data[0 len 4]
    gas gas_remaining wei
    args call.data[4 len calldata.size - 4]
```

```
# Pattern #2
def _fallback() payable:
  delegate 0x2f5[...]7adb5 with:
    funct call.data[return_data.size len 4]
    gas gas_remaining wei
    args call.data[return_data.size + 4 len calldata.size - 4]
```

```
# Pattern #3
def unknown9ddbbf85(uint256 _param1, uint256 _param2) payable:
  delegate unknownf8a3176cAddress.0x9ddbbf85 with:
    gas gas_remaining - 10000 wei
    args _param1, _param2
```

Figure 6.5: Three bytecode features illustrating the two conceptual properties of proxy contracts (Section 2.15.4) at the decompiled bytecode level.

Using the extracted features from the above study, we then scan the proxy's decompiled bytecode to locate all relevant `delegatecall` statements. This allows us to extract the specific variable that holds the implementation contract address (A5). After analyzing the decompiled bytecode of a proxy contract, if no relevant `delegatecall` statements (i.e., proxy functionality) are found, we halt the

entire detection process with a "Failure: Delegate Not Found" flag and label the proxy contract as Non-UPC (A6). Otherwise, we iterate over the set of all relevant `delegatecall` statements, examining each one individually through steps A8 to A15. The control flow of this loop is governed by a conditional check at step A7, labeled "Last relevant `delegatecall` statement?" This condition ensures the loop terminates after processing the final relevant `delegatecall`, thereby preventing infinite iteration. Once all `delegatecall` statements have been evaluated, we proceed to step A16, where the proxy is classified as either UPC or Non-UPC based on the information stored in global memory (A7). This global memory functions as a shared scratchpad for recording information relevant to all three detectors: A: SMUP, B: ESUP, and C: DUP. It can be updated by any of these detectors via the "Upgrade Function Detector" component at steps A13, B13, and C9, respectively.

Specifically, for each relevant `delegatecall`, the detector records the signature of the identified upgrade function, its line number, and the address of the contract in which the upgrade function is located, all stored in the global memory. For each relevant `delegatecall`, we first determine how the implementation contract address is obtained. There are two possibilities. First, the address may be retrieved through an external call to another contract that holds the implementation address, this corresponds to variant #1 and variant #2 of the ESUP design, as shown in Figure 6.3. Second, the address may be stored directly in the proxy's storage, as in the SMUP design. In this case, the corresponding variable definition, referred to as the `implementation variable`, can be found in the proxy's storage layout.

In the former case (external storage), the process proceeds to step A9. In the latter case (internal storage), we continue with steps A10–A15 to examine whether

the `implementation variable` is updated within the proxy or its implementation contracts (A8). In cases where the implementation address is retrieved through an external call, we invoke the "ESUP Detector" component to analyze the proxy's external dependencies for signs of upgradeability. After executing this component for the current relevant `delegatecall`, the process returns to step A7 to examine the next relevant `delegatecall`.

In cases where the implementation variable is stored within the proxy, we parse the `delegatecall` statement to extract this variable (e.g., the `imp` variable in the SMUP reference design in Figure 6.3). This variable holds the address of the implementation contract. If it is reassigned within a public or external function (e.g., `upgradeTo`), it signifies an update to the implementation contract address, thereby indicating an upgrade. We refer to this variable as the `primary impact variable`. Subsequently, we extract the storage slot of the primary impact variable (also known as the implementation slot) (A10). It is not always the case that the primary impact variable is directly updated by a function parameter at runtime. For example, developers may first assign the new implementation address to a pending implementation variable (e.g., as a proposal). Upon approval, typically via a governance or voting mechanism, this pending variable then replaces the primary impact variable. As a result, the primary impact variable may be indirectly modified through one or more intermediary variables. These variables, which have the potential to trigger upgrades, are referred to as "auxiliary impact variables".

To identify such variables, the primary impact variable and its corresponding storage slot are passed to the "Auxiliary Impact Variable Detector" component. Both the primary and auxiliary variables are collectively referred to as *impact variables*,

as changes to any of them may result in an upgrade of the implementation contract (A11). We then process each impact variable (A12). For each one, we examine all assignment operations within the proxy contract to determine whether its value is modified within a function. This analysis is performed by the "Upgrade Function Detector" component. If an upgrade function is identified, its function signature is recorded. The process then continues with the next impact variable to detect all potential upgrade functions (A13).

After processing all impact variables, we check whether at least one upgrade function was identified that modifies the value of any of these variables (A14). If so, we return to step A7 to evaluate the next `delegatecall` statement. If no such function is found, it is possible that the impact variables are modified within one of the proxy's implementation contracts. In this case, we proceed to step A15, where we analyze whether the proxy conforms to the DUP reference design by examining its implementation contracts using the "DUP Detector" component. After this analysis, we return to step A7 to process the next relevant `delegatecall` statement. Once all relevant `delegatecall` statements have been evaluated, we inspect the global memory. If at least one `delegatecall` statement is associated with an upgrade function, identified by any of the detectors, we classify the proxy as a UPC. Otherwise, it is labeled as a Non-UPC (A16).

6.2.4.2.2 ESUP Detector. This component identifies UPCs that follow the Externally Supervised Upgradeable Proxy (ESUP) reference design (variants #1 and #2 in Figure 6.3). Specifically, if the address of the implementation contract is retrieved through a call to an external contract, referred to as a target dependency, prior to the `delegatecall` operation, then that dependency may contain the upgradeability

mechanism. The primary goal of this component is to analyze the proxy's dependencies for signs of upgradeability. The component takes as input the proxy contract and its associated implementation contracts. Figure 6.6 illustrates the algorithm.

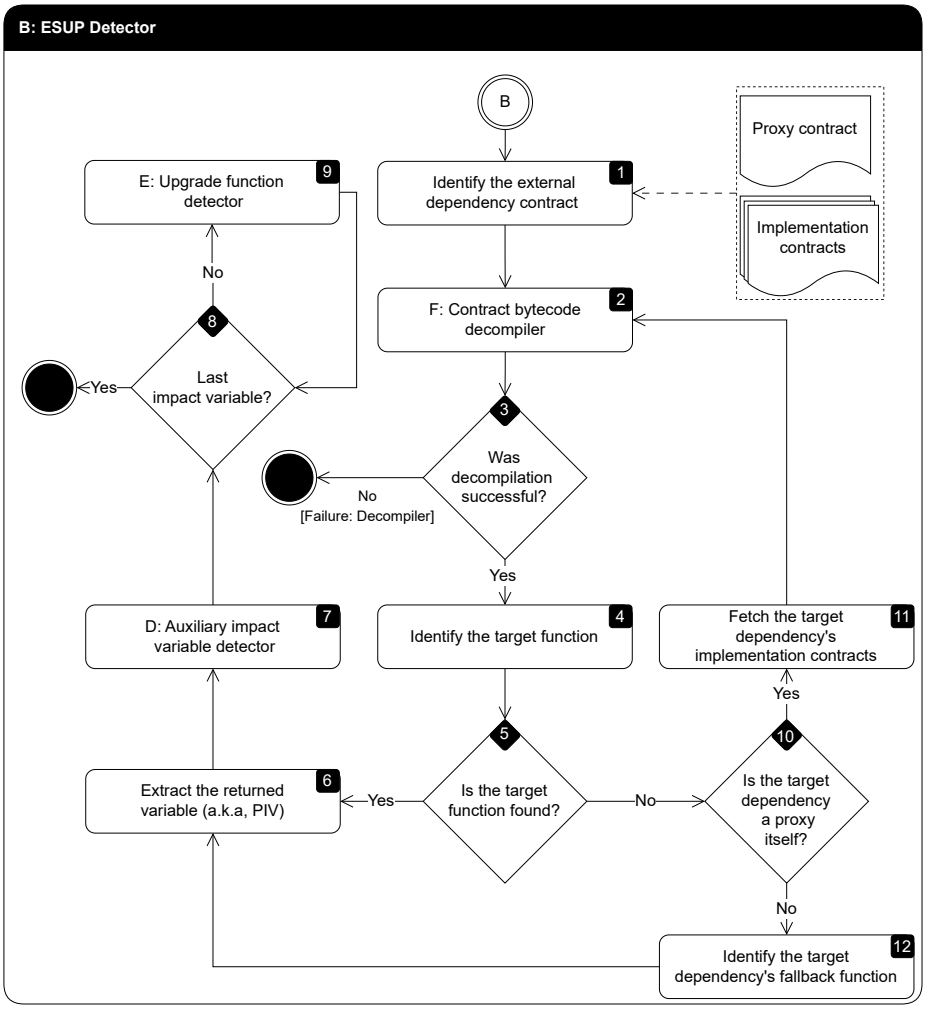


Figure 6.6: Detailed design of the "ESUP Detector" component.

The first step is to detect the target dependency, as a proxy may have multiple external dependencies. The target dependency is a contract that stores and

returns the address of the implementation contract. As previously noted, prior to executing a `delegatecall`, the proxy typically makes an external call (e.g., to a `getImplementation` function) to the target dependency contract to retrieve this address. To identify the target dependency, we perform a dynamic analysis. Specifically, we examine the proxy's `delegatecall` transactions and locate the trace that immediately precedes the `delegatecall` trace and returns the implementation contract address used in the subsequent `delegatecall`. The recipient of this preceding trace is the target dependency contract (B1). We then collect and decompile the bytecode of the target dependency contract (B2). If Panoramix fails to generate a decompiled source file, we halt the detection process with a "Failure: Decompiler" flag and return to step A7 of the "SMUP Detector" component to continue with the next relevant `delegatecall` statement (B3).

Once the target dependency's bytecode is successfully decompiled, we search for the target function (e.g., `getImplementation`) that returns the implementation contract address. The value returned by this function reveals the implementation variable within the target dependency contract. To identify this function, we use the `calldata` of the target trace. Specifically, the first four bytes of the `calldata` indicate the function selector of the target function invoked on the target dependency contract. We then extract all functions from the decompiled bytecode of the target dependency and compute their corresponding function selectors (B4).

We match the selector of the target function against all function selectors extracted from the target dependency contract to determine whether the target function is implemented within the contract. If a match is found, we proceed to step

B6; otherwise, we skip to step B11 (B5). In step B6, we extract the variable returned by the target function, referred to as the primary impact variable. Next, we identify any auxiliary impact variables associated with the primary variable (B7). We then evaluate each impact variable in step B9. After all impact variables have been processed, we return to step A7 of the "SMUP Detector" component (B8). For each impact variable, we examine all assignment operations within the target dependency contract to determine whether an upgrade occurs. If an upgrade function is detected, we record the signature of the function where the update takes place and continue with the next impact variable (B9).

If no match is found for the target function within the target dependency contract, there are two possible explanations. First, the target dependency contract may itself be a proxy that redirects calls to its own implementation contract, where the target function is actually defined, this corresponds to variant #2 of the ESUP design in Figure 6.3. Alternatively, if the target dependency is not a proxy, then the target function is likely the `fallback` function of the target dependency contract, since the `fallback` function is triggered when no matching function signature is found within the contract (B10). As outlined in variant #2 of the ESUP reference design, if the target dependency is identified as a proxy, we retrieve its implementation contracts and recursively repeat the process starting from step B2, this time analyzing the target dependency's implementation contracts (B11). If the target dependency is not a proxy, we extract its `fallback` function and treat it as the target function, then proceed from step B6 (B12).

6.2.4.2.3 DUP Detector. This component detects UPCs that follow the Delegated Upgradeable Proxy (DUP) reference design. Specifically, if no upgrade function is found for the set of impact variables within the proxy contract itself (as determined in step A12), we must analyze the proxy's implementation contracts to determine whether any of them implement a function capable of updating one of these impact variables. The inputs to this component are: (i) the proxy's implementation contracts, (ii) the primary impact variable, and (iii) the complete set of impact variables along with their corresponding storage slots, as identified in steps A10 and A11. Figure 6.7 illustrates the algorithm.

More specifically, we begin by examining the data type of the primary impact variable found in the proxy contract. If it is not a mapping, we proceed to step C4. Otherwise, the proxy may follow the ERC-2535 Diamonds UPC Pattern, corresponding to variant #3 of the DUP reference design in Figure 6.3, and we continue to step C2. In step C2, we analyze three well-known implementations of the Diamond proxy pattern (Mudge 2022), namely `diamond-1-hardhat`, `diamond-2-hardhat`, and `diamond-3-hardhat`. From these, we extract the `diamondCut` function signatures, which collectively yield a single unique function signature[1]. Next, using an Infura Ethereum node, we call the DiamondLoupe's `facetAddress` function with the `diamondCut` function selector as input (C2). If the call returns a facet address other than `0x0`, the proxy follows the Diamond UPC Pattern, and we return to the "SMUP Detector" process. Otherwise, we continue to step C4 (C3). In step C4, we analyze the proxy's implementation contracts one by one.

[1] `diamondCut((address,uint8,bytes4[])[],address,bytes)`

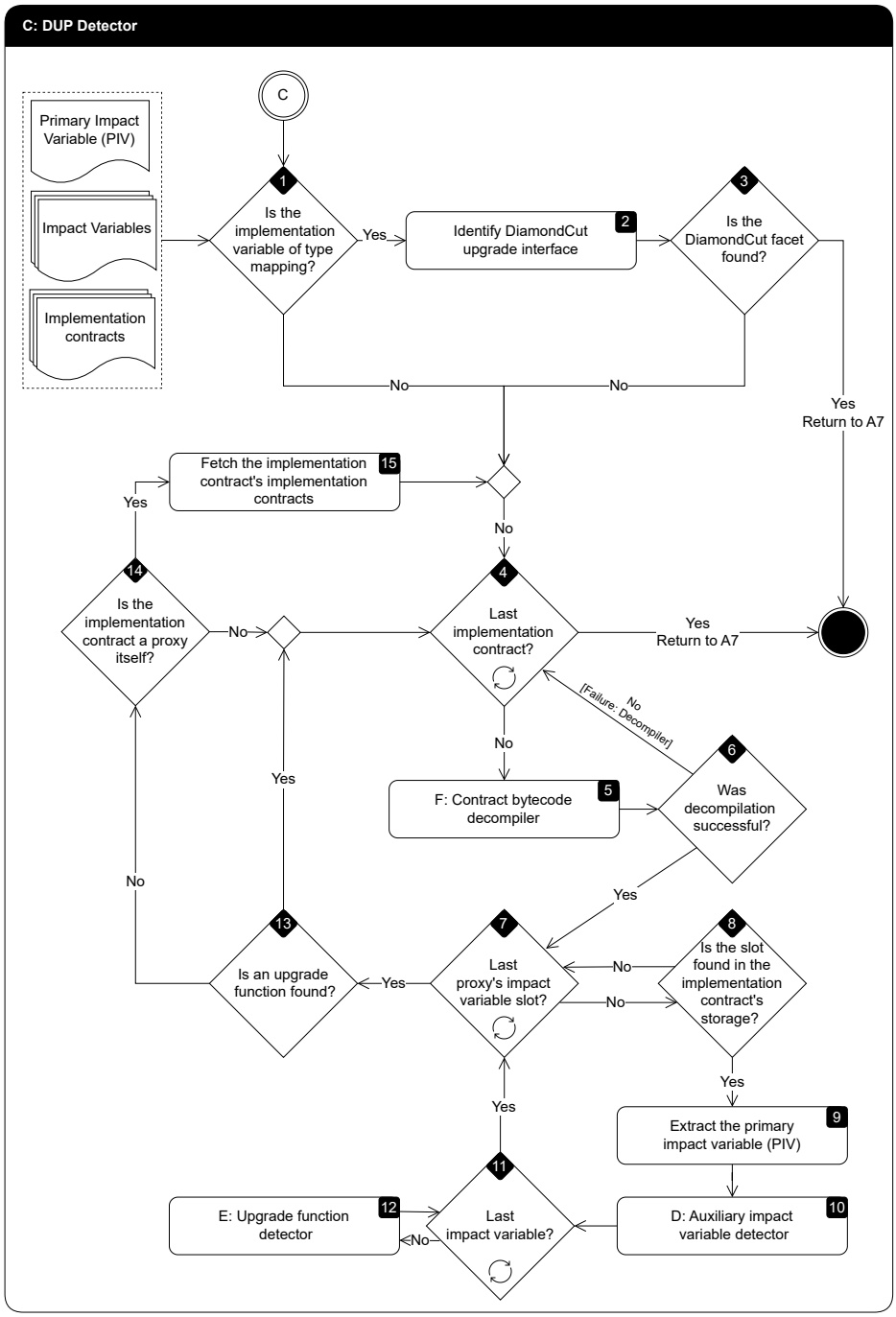


Figure 6.7: Detailed design of the “DUP Detector” component. Small diamonds without decision criteria represent UML flowchart merge nodes.

After processing all implementation contracts, we return to the "SMUP Detector" process (C4). For each implementation contract associated with the proxy, we retrieve and decompile its bytecode (C5). If Panoramix fails to generate a decompiled source file, we halt the analysis for that specific implementation contract with a "Failure: Decompiler" flag and continue to the next one (C6). If decompilation is successful, we proceed to match each input impact variable's storage slot against the storage layout of the implementation contract. Specifically, for each impact variable's storage slot in the proxy contract (C7), we search for a corresponding slot in the implementation contract's storage (C8). If a match is found, we record the variable's name and storage slot, now considered the primary impact variable for the implementation contract (C9). If no match is found, we proceed to the next impact variable's slot. Steps C10, C11, and C12 mirror steps A11, A12, and A13, respectively. After analyzing all impact variables, if at least one upgrade function is identified, we record it and move to the next implementation contract. If no such function is found, we proceed to step C14 (C13).

The reason we may not find an upgrade function within an implementation contract could be that the implementation itself is a proxy, and the upgrade function is actually implemented in one of its own implementation contracts, corresponding to variant #2 of the DUP reference design. If this is the case, we first retrieve the implementation contract's implementation contracts (A15) and recursively repeat the analysis for each, starting from step A4. If the implementation contract is not a proxy, we simply proceed to the next implementation contract (C14).

6.2.4.2.4 Auxiliary Impact Variable Detector. Once the primary impact variable is identified, we proceed to detect additional auxiliary variables that may influence the value of the primary implementation variable. Figure 6.8 illustrates our algorithm. We classify these auxiliary variables into two categories: secondary and tertiary impact variables. A variable is considered a secondary impact variable if its storage slot is similar to that of the primary impact variable. For each such variable, we extract its name and corresponding storage slot. To identify secondary impact variables, we examine all storage variables based on this condition (D1). A variable is considered a tertiary impact variable if it is assigned to either a primary or secondary impact variable and is also found in the contract's storage. For each such variable, we again extract its name and storage slot. To identify tertiary impact variables, we evaluate all assignment expressions based on the above conditions (D2).

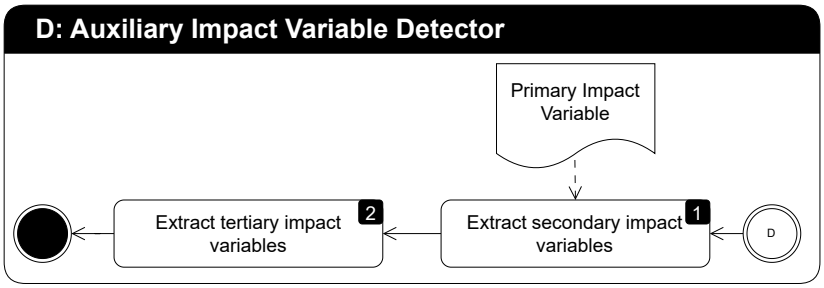


Figure 6.8: Detailed design of the "Auxiliary Impact Variable Detector" module.

6.2.4.2.5 Upgrade Function Detector. The purpose of this module is to identify functions in which the value of an impact variable is reassigned to a new value. The inputs to this module are an impact variable and the decompiled source code of

a contract. Figure 6.9 illustrates the algorithm. We begin by retrieving all assignment statements where the left-hand side operand (i.e., lvalue) matches the impact variable (E1). Each assignment is then processed individually (E2). For a given assignment, we extract the right-hand side operand (i.e., rvalue) (E3) and examine its origin. If the rvalue is either (i) a parameter of the function in which the assignment occurs or (ii) dynamically obtained via a call to another contract, we classify the enclosing function as an upgrade function. Otherwise, we continue to the next assignment (E4). For each detected upgrade function, we store its function signature, the line number where the assignment occurs, and the contract address in which it is defined, all within the global memory. We then proceed to analyze the next assignment to identify all potential upgrade functions (E5).

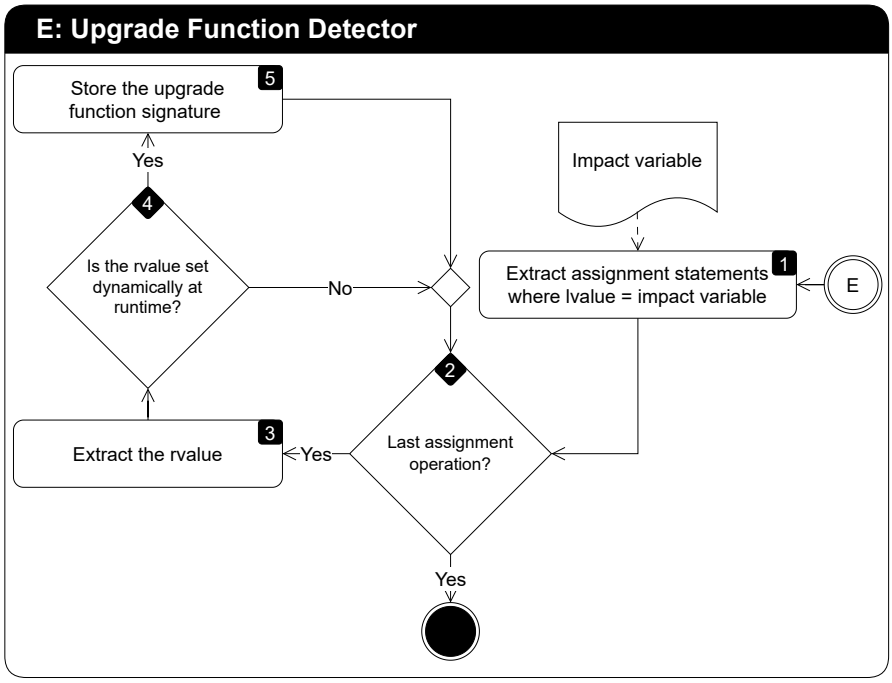


Figure 6.9: Detailed design of the “Upgrade Function Detectors” module.

6.2.4.2.6 Contract Bytecode Decompiler. The objective of this module is to collect and decompile the bytecode of a given contract (Figure 6.10). The input to this module is a contract address. Using this address, we query the contracts table from the BigQuery Ethereum public dataset to retrieve the bytecode associated with the contract (F1). Once the bytecode is obtained, we use the Panoramix decompiler (Section 2.13) to generate a decompiled version of the contract's source code (F2).

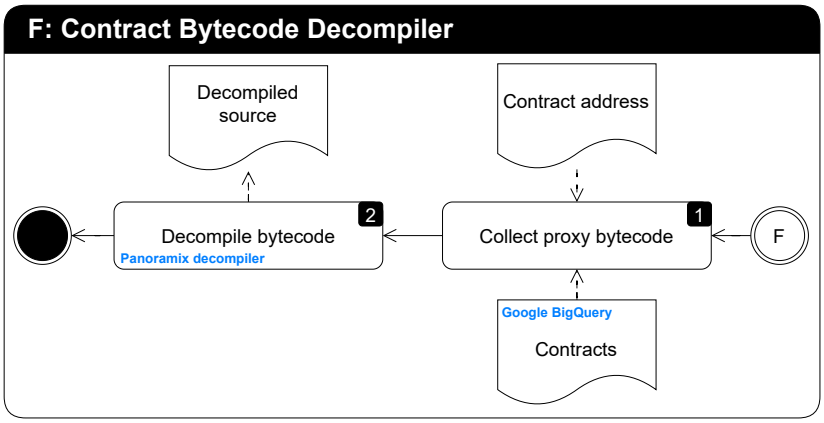


Figure 6.10: Detailed design of the "Contract Bytecode Decompiler" module.

6.2.5 Upgradeable Proxy Contract Pattern Classifier Layer

Our Proxy Contract Type Classifier Layer employs both static and dynamic techniques to determine whether a given contract is an UPC or Non-UPC (a.k.a., FPC). While this layer initially identifies UPCs at a high level by determining the UPC Reference Design they follow, UPC Sentinel further classifies each UPC by detecting the specific UPC Pattern used in its implementation, such as the Transparent UPC Pattern, the ERC-1822 Universal UPC Pattern, or the Beacon UPC Pattern. This fine-grained classification provides added value in several key areas.

First, understanding the specific UPC Pattern enhances transparency by allowing users and stakeholders to verify how upgrades are managed and what control mechanisms are in place, thereby fostering trust. Second, identifying the pattern enables automated analysis tools to apply targeted validation logic, improving the efficiency and accuracy of contract assessments. Finally, recognizing the UPC Pattern facilitates compliance verification by ensuring that UPC implementations conform to established industry standards, thereby supporting best practices and regulatory adherence.

As such, this additional Pattern Classifier Layer provides a fine-grained classification of detected UPCs by identifying the specific type of UPC Pattern each instance follows, based on its decompiled bytecode. In prior work, Bodell III et al. (2023) characterized the syntactic features of various UPC Patterns at the source code level. While these syntactic features help reveal how different UPC Patterns manifest in source code, the lower-level nature of bytecode and the linearization effects introduced during compilation necessitate a re-examination to determine whether, and how, UPC Patterns are preserved or transformed in the decompiled bytecode. To address this, we also leveraged the formal specifications of UPC Patterns to identify key distinguishing features that enable their detection at the decompiled bytecode level.

Furthermore, some storage-related information may be lost at the decompiled bytecode level. Specifically, bytecode does not explicitly encode the contract's storage layout; instead, decompilers like Panoramix must infer this layout by analyzing `SSTORE` (write) and `SLOAD` (read) instructions. As a result, if a variable, particularly one with `internal` visibility, is never involved in any computation (i.e., it is neither

read from nor written to), the decompiler cannot detect its presence based solely on bytecode instructions. Consequently, such a variable will not be reflected in the inferred storage layout of the decompiled contract. Nonetheless, an accurate storage layout at the decompiled bytecode level is critical for identifying certain patterns, such as Inherited and Eternal Upgradeable Proxy contracts, where consistency between the proxy and implementation contract storage layouts is a key factor for correct classification.

Figure 6.11 outlines the decision-making process used to determine the specific UPC Pattern that a given UPC instance follows. When classifying the UPC Pattern of a UPC, we adopt a systematic approach. First, the Proxy Contract Type Classifier Layer determines both the upgradeability status and the reference design of the contract. Then, within the identified reference design, we determine the most specific UPC Pattern to which the UPC conforms. A Non-pattern category is also included to capture UPCs that do not align with any recognized patterns or whose specific type cannot be determined.

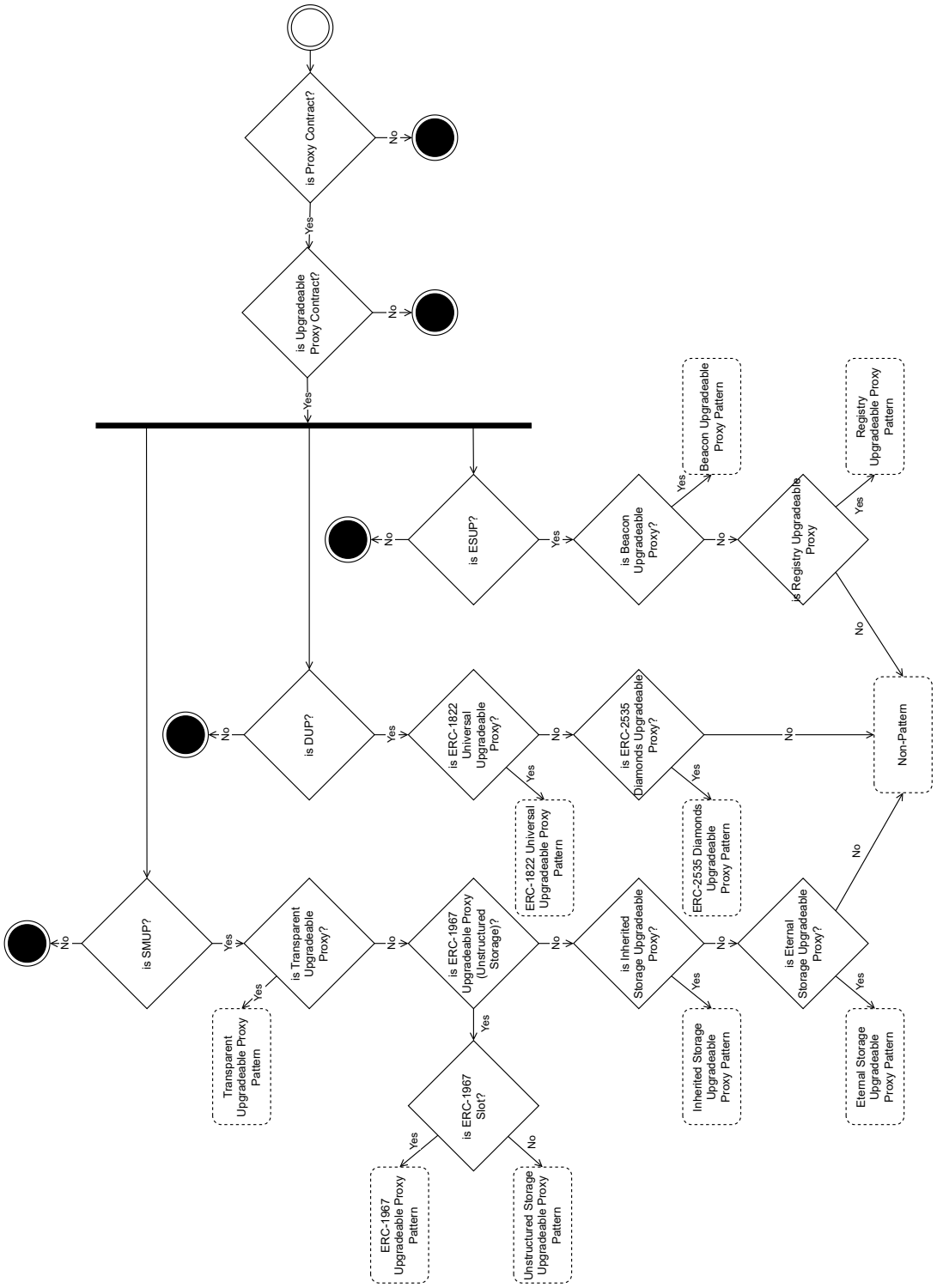


Figure 6.11: A systematic top-down approach for identifying the type of upgradeable proxy pattern a UPC adheres to.

More specifically, based on our three UPC Reference Designs (Section 6.2.4.1), a UPC can be classified under only one reference design at a time, either SMUP, DUP, or ESUP. This exclusivity arises because each reference design is grounded in a distinct upgradeability mechanism (Section 6.2.4.1). Similarly, at the UPC Pattern level, a proxy cannot simultaneously belong to patterns associated with different reference designs. For instance, a proxy that follows the ERC-1822 Universal UPC Pattern cannot also be classified under the Inherited Storage UPC Pattern. This restriction stems from the fundamental differences in the underlying upgradeability mechanisms across reference designs: in the former, the implementation contract is responsible for executing the upgrade logic, whereas in the latter, the proxy itself directly implements the upgrade functions.

Furthermore, among the four patterns in the SMUP reference design, three, the Inherited Storage Upgradeable Proxy, the Eternal Storage Upgradeable Proxy, and the ERC-1967 Upgradeable Proxy (Unstructured), introduce fundamental principles for designing the storage layout of both the proxy and its implementation contracts to minimize the risk of storage collisions (OpenZeppelin 2018a; Bodell III et al. 2023). Storage collisions occur when the proxy and its implementation contracts use the same storage slots for different purposes, potentially leading to the corruption of critical data such as the admin address or the implementation contract address. Notably, storage collisions can arise in any proxy contract, regardless of the specific UPC Pattern employed.

Although the three aforementioned UPC Patterns were originally developed for designing SMUP-based UPCs, their storage layout principles, namely, Inherited Storage, Eternal Storage, and ERC-1967 (Unstructured) Storage, are widely adopted

across other patterns to prevent storage collisions. For example, any of these three principles can be used to design the storage layout of the Transparent UPC Pattern, which is the fourth pattern within the SMUP reference design. In such cases, we classify the UPC under the most specific UPC Pattern, i.e., the Transparent UPC Pattern, because it introduces an additional distinguishing feature: a mechanism for controlling who is authorized to call the proxy functions. Similarly, ERC-1967 (Unstructured) Storage is commonly used in other proxy patterns such as the ERC-1822 Universal UPC Pattern and the Beacon UPC Pattern. Therefore, the adoption of these storage layout principles in other UPC Patterns does not constitute a new reference design or UPC Pattern; rather, it reflects the application of established best practices to prevent storage collisions (Section 2.15.2).

In the following sections, we describe the specific methods used to detect each pattern shown in Figure 6.11, within the context of its corresponding UPC Reference Design.

6.2.5.1 SMUP Pattern Classifier

For any detected UPC that conforms to the SMUP reference design, we evaluate it against the following patterns in the specified order: Transparent UPC Pattern, ERC-1967 UPC Pattern (Unstructured Storage), Inherited Storage UPC Pattern, and Eternal Storage UPC Pattern. If the UPC does not match any of these known patterns, it is categorized under the Non-pattern category. This typically occurs when the proxy's storage layout deviates from recognized standards.

6.2.5.1.1 Transparent UPC Pattern. This pattern resolves the issue of function clashing between a proxy and its logic contract by routing calls based on the caller's

```
1  def storage:
2    stor3608 is addr at storage
         0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc
3  [...]
4  def _fallback() payable: # default function
5    if caller == addr(storB531.field_0):
6      revert with 0x8c[...]00, 32, 66, 0x74[...]65, mem[230 len 30]
7    delegate uint256(stor3608.field_0) with:
8      funct call.data[0 len 4]
9      gas gas_remaining wei
10     args call.data[4 len calldata.size - 4]
11   if not delegate.return_code:
12     revert with ext_call.return_data[0 len return_data.size]
13   return ext_call.return_data[0 len return_data.size]
14 [...]
```

Listing 6.1: Example demonstrating how the Transparent UPC Pattern routes calls based on the caller's address (see lines 5–6).

address. If the caller is the proxy's admin, the proxy handles the call internally; otherwise, it delegates the call to the logic contract (Palladino 2021; Bodell III et al. 2023). This is a key aspect that distinguishes this pattern compared to Inherited, Eternal and ERC-1967 Upgradeable Proxy (Unstructured Storage) Patterns. Listing 6.1 shows this pattern at decompiled bytecode level for a contract selected from wild. To detect this, UPC sentinel analyzes the decompiled bytecode of the detected proxy function from its definition (Line 4) up to where the `delegatecall` is called (Line 7) to assess whether there is a conditional check (Line 5) on the caller of the proxy function before delegation. If we find one, we flag the UPC as an instance of the Transparent UPC Pattern. Otherwise, the contract does not conform to this pattern.

6.2.5.1.2 ERC-1967 UPC Pattern. In this pattern, the implementation address is stored in a dedicated, fixed storage slot within the proxy contract, unlike the Inherited or Eternal UPC Patterns, where storage slots are allocated sequentially. This dedicated slot is derived using a hashing mechanism to minimize the risk of collisions with the state variables of the logic contract. UPC Sentinel identifies this pattern by examining the storage slot associated with the implementation variable (i.e., the variable that stores the implementation contract address). If the storage slot is represented as a long hexadecimal string (i.e., a 32-byte slot), the UPC is classified as an instance of this pattern. Building on this detection, we further refine the classification by dividing this pattern into two subcategories.

The concept of using a hashing mechanism is a shared foundation between Unstructured Storage (OpenZeppelin 2018e) and ERC-1967 Storage (Santiago Palladino 2019b). However, the latter represents an official Ethereum proposal that specifies a standardized slot for the implementation contract address. Given the significance of ERC-1967, we divide this category into two subtypes: i) ERC-1967 UPC Pattern, identified when the implementation variable uses the exact slot defined by the ERC-1967 standard; and ii) Unstructured Storage UPC Pattern, identified when a hashed slot other than the ERC-1967 standard is used. Listing 6.1 illustrates an example of an ERC-1967 storage slot used to store the implementation variable (`stor3608`), explicitly declared on line 2 following the storage keyword.

6.2.5.1.3 Inherited Storage UPC Pattern. This pattern prevents storage clashes by introducing a shared storage structure that is inherited by both the proxy and the implementation contracts. Specifically, a dedicated contract defines and maintains the state variables required for upgradeability (e.g., the admin address or the

implementation contract address). By ensuring that both the proxy and the implementation contract inherit this shared storage structure as the first base contract, consistent alignment of the storage layout is achieved. In addition to the shared storage, the implementation contract may define its own state variables. However, to avoid conflicts, these variables must be declared after the inheritance of the shared storage (OpenZeppelin 2018d). This guarantees that the new storage slots defined by the implementation contract do not overlap with those used by the proxy.

At the decompiled bytecode level, where inheritance relationships are flattened, detection requires comparing the proxy's storage layout with that of its implementation contract. Specifically, we must observe that the proxy's storage layout is a consistent subset of the implementation contract's layout: all state variables in the proxy must appear at identical storage slots in the implementation contract, while any additional variables in the implementation contract must occupy subsequent, non-overlapping slots. If these conditions are met, we classify the UPC as an instance of the Inherited Storage UPC Pattern; otherwise, it is not considered to conform to this pattern. For example, lines 2 to 4 in Listing 6.2 and Listing 6.3 illustrate how the three upgradeability-related state variables from the proxy are consistently preserved in the implementation contract's bytecode, thereby demonstrating proper alignment of shared storage and correct allocation of additional variables in distinct, non-overlapping slots.

6.2.5.1.4 Eternal Storage UPC Pattern. This pattern ensures consistent and immutable storage structures by centralizing all state variables, primarily predefined internal mapping variables (e.g., for integers, addresses, and strings), within a

```
def storage:
  owner is addr at storage 0
  adminAddress is addr at storage 1
  implementationAddress is addr at storage 2

[...]
```

Listing 6.2: Example storage layout of the proxy contract in the Inherited Storage UPC Pattern.

```
def storage:
  owner is addr at storage 0
  adminAddress is addr at storage 1
  implementationAddress is addr at storage 2
  stor3 is mapping of uint8 at storage 3
  stor4 is mapping of uint8 at storage 4
  stor5 is mapping of uint8 at storage 5
  [...]
[...]
```

Listing 6.3: Example storage layout of the implementation contract in the Inherited Storage UPC pattern.

generic `EternalStorage` contract. The implementation contracts of a proxy inherit from this centralized storage contract, thereby ensuring uniform access to shared state variables. The proxy contract must also inherit from `EternalStorage` as its *first* base contract to maintain alignment in storage layout with its implementation contracts. Following this, the proxy typically inherits from another contract that introduces upgradeability-specific state variables, such as the admin address and the implementation contract address (OpenZeppelin 2018c).

This pattern exhibits several distinct characteristics (OpenZeppelin 2018c). First, both the proxy and its implementations inherit from `EternalStorage` as the first base contract, ensuring consistent storage slot usage for all mapping-type variables.

Second, the proxy introduces additional storage slots for upgradeability-related variables, which are not shared with the implementation. Third, the implementation contract preserves the immutability of the storage layout by avoiding the declaration of new state variables. Unlike the Inherited Storage UPC Pattern, where the proxy's layout is a subset of the implementation's, here, the implementation's layout is instead a subset of the proxy's.

A key feature of this pattern is its clear separation of concerns: implementation contracts are responsible for managing the mappings within `EternalStorage`, while the proxy focuses on securely delegating function calls to enable these updates. Notably, the proxy typically does not directly access the mapping variables (e.g., via read or write operations); instead, such operations are delegated to the implementation contracts. In Solidity, storage slots are assigned sequentially to declared state variables, enabling predictable layouts for future upgrades. However, at the decompiled bytecode level, this layout is reconstructed by analyzing opcodes such as `SSTORE` and `SLOAD`. If an internal mapping variable is declared but never referenced in computations or logic, its corresponding opcodes do not appear in the bytecode. Consequently, decompilers cannot infer its existence, even though it occupies a storage slot.

This limitation poses a challenge in identifying this pattern at the decompiled bytecode level. Listings 6.4 and 6.5 illustrate an example of this pattern's storage layout in both the proxy and implementation contracts. In this example, both contracts inherit from `EternalStorage`, which defines six mapping variables. As a result, the compiler allocates the first six storage slots to these mappings. However, in the proxy contract, the first recognized slot in the decompiled output is slot 6,

```
def storage:
  upgradeabilityOwner is addr at storage 6
  version is uint256 at storage 7
  implementationAddress is addr at storage 8

[...]
```

Listing 6.4: Example storage layout of the proxy contract in the Eternal Storage UPC Pattern.

```
def storage:
  deployedABlock is mapping of uint256 at storage 0
  unknown871c0760 is mapping of addr at storage 2
  stor4 is mapping of uint8 at storage 4

[...]
```

Listing 6.5: Example storage layout of the implementation contract in the Eternal Storage UPC Pattern.

since slots 0 through 5 are associated with mappings that are not referenced in the code and thus omitted by the decompiler. Similarly, in the implementation contract (Listing 6.5), mappings appear at slots 0, 2, and 4, while slots 1, 3, and 5 remain unrecognized. This lack of visibility complicates the task of verifying storage layout consistency between proxy and implementation contracts, an essential aspect for correctly identifying this pattern.

To address this issue, we rely on the design principles underlying the storage layout of this pattern (OpenZeppelin 2018c). Specifically, as previously discussed, after inheriting from `EternalStorage`, the proxy must also inherit from another contract that declares upgradeability-related state variables (OpenZeppelin 2018c). These variables are typically inferred in the proxy's decompiled storage layout, as

they are actively used by the proxy (lines 2 to 4), and they occupy the slots immediately following those allocated to the `EternalStorage` mappings. For instance, in Listing 6.4, the variable `upgradeabilityOwner` begins at slot 6, directly after the last mapping variable, which resides in slot 5 but is not inferred in the decompiled bytecode.

To detect this pattern, we first scan the proxy's storage layout and identify the first variable whose slot is represented as an integer. We refer to this as the *indicator slot*, which serves as a key marker. If the indicator slot is greater than zero (e.g., slot 6 on line 2), it suggests that the preceding slots are likely occupied by internal `EternalStorage` mapping variables that were not inferred by the decompiler. Given that the implementation contract must also inherit from `EternalStorage` and utilize its mappings, we then examine its storage layout. If all inferred variables with slots less than the indicator slot are of type `mapping`, this implies that the implementation contract also inherits from `EternalStorage`, and that the indicator slot marks the beginning of upgradeability-related state variables in the proxy. In this case, we classify the contract as an instance of the *Eternal Storage Upgradeability Proxy Pattern*. If these conditions are not met, the UPC is considered not to conform to this pattern and is instead categorized under the Non-pattern class.

6.2.5.2 DUP Pattern Classifier

For any detected UPC that follows the DUP reference design, we further classify it into one of the following categories: ERC-1822 Universal UPC Pattern and ERC-2535 Diamonds Upgradeability Proxy Pattern.

6.2.5.2.1 ERC-1822 Universal UPC Pattern. According to the ERC-1822 standard, the proxy stores the implementation contract address in a specific, predefined fixed storage slot, calculated in an unstructured manner. Additionally, every implementation contract must inherit from a contract known as `Proxiable` (Bodell III et al. 2023; Gabriel Barros 2019b), which defines two key functions. The first, `updateCodeAddress`, is an internal function that safely updates the implementation contract address. The second, `proxiableUUID`, is a public or external `view` function that returns the storage slot used to store the implementation address. A key distinguishing feature of this pattern is the compatibility check performed during an upgrade via the `updateCodeAddress` function. Specifically, before updating the implementation address, `updateCodeAddress` verifies whether the new implementation contract supports the `Proxiable` interface and uses the same storage slot. This compatibility check is performed through a call to `proxiableUUID` on the new implementation contract, ensuring that the upgrade adheres to the required storage structure and maintains consistency.

Detecting this pattern at the decompiled bytecode level requires a different approach. In decompiled bytecode, inheritance relationships are flattened into a single, continuous sequence of operations, obscuring the original contract structure and hierarchy. Therefore, we cannot directly verify whether the implementation contract inherits from `Proxiable`. However, according to the standard (Gabriel Barros 2019b), the public upgrade function in the implementation contract must internally call the `updateCodeAddress` function as part of its logic. Since `updateCodeAddress` is an internal function, its logic is inlined into the public upgrade function during

```
1  def unknown912a9885(addr _param1) payable:
2    [...]
3    if not caller:
4      revert with [...]
5    [...]
6    static call _param1.0x52d1902d with:
7      gas gas_remaining wei
8    [...]
9    if ext_call.return_data[0] !=
         0xc5f16f0fcc639fa48a6947836d9850f504798523bf8c9a3a87d5876cf622bcf7:
10     revert with 0, 'Not compatible'
11   unknownabd108baAddress = _param1
12   [...]
```

Listing 6.6: Excerpt of the `upgrade` function from the implementation contract associated with an ERC-1822 Universal UPC Pattern.

compilation. Thus, at the bytecode level, we can detect this pattern by analyzing whether the public upgrade function includes the logic corresponding to the `proxiableUUID` compatibility check. Listing 6.6 presents an excerpt of the upgrade function from the implementation contract associated with a proxy of this type.

As shown, the function accepts a new address as input (line 2). To verify compatibility, the upgrade function calls the `proxiableUUID` function, identified by the selector `0x52d1902d`, on the provided address (line 7). It then compares the returned storage slot with the predefined fixed storage slot used for the implementation contract address (line 10). If both of these checks are present within the upgrade function of the implementation contract, we flag the UPC as an instance of the ERC-1822 UPC Pattern. Otherwise, the contract is considered not to conform to this pattern. Additionally, details such as the location of the upgrade function(s) in the decompiled bytecode and the specific storage slot used for the implementation contract address are precomputed by UPC Sentinel's upgradeability detector layer

during the UPC detection phase. These details are stored in the global memory scratchpad (Section 6.2.4.2) to support the detection of this pattern.

6.2.5.2.2 ERC-2535 Diamonds UPC Pattern. We classify a UPC as an instance of this pattern if it precisely aligns with one of the three official implementations, `diamond-1-hardhat`, `diamond-2-hardhat`, or `diamond-3-hardhat` (steps C1 to C3 in Figure 6.7) (Mudge 2018b). If the UPC does not conform to any of these reference implementations, it is categorized under the Non-pattern class.

6.2.5.3 ESUP Pattern Classifier

For any detected UPC that follows the ESUP reference design, we further classify it into one of the following categories: Beacon UPC Pattern and Registry UPC Pattern.

6.2.5.3.1 Beacon UPC Pattern. This pattern relies on a defined standard where the `BeaconProxy` interacts with its `Beacon` contract through a specific interface. This interface includes a function implemented by the `Beacon` contract that returns the address of the implementation contract it stores. In our approach, UPC Sentinel's ESUP detector component verifies the presence of this function in the `Beacon` contract. As a result, any UPC conforming to the ESUP reference design inherently satisfies this condition. Additionally, the standard mandates that the `Beacon` contract stores the address of a *single* implementation contract. This serves as the key distinguishing feature compared to the Registry UPC Pattern (Dharma 2019). To confirm this, we analyze the type of the implementation variable (i.e., the variable

holding the implementation contract address) returned by the `Beacon`. If the variable is of a type that stores a single address, we classify the UPC as an instance of this pattern. Otherwise, the contract is not considered to conform to this pattern.

6.2.5.3.2 Registry UPC Pattern. This pattern is very similar in mechanism to the Beacon UPC Pattern, with one major difference: it allows tracking of multiple versions of the implementation contract address (Diligence 2017; Bodell III et al. 2023). In other words, the `Registry` contract can store multiple addresses, whereas the `Beacon` contract is limited to a single address. Therefore, we flag the UPC as an instance of this pattern if the implementation variable exhibits *multiplicity* (e.g., an array or mapping of addresses or objects) (Bodell III et al. 2023; Diligence 2017). This multiplicity indicates that the contract maintains a record of past and present implementation addresses, thereby enabling a versioning mechanism. If this condition is not met, the UPC is classified under the Non-pattern category.

6.3 Data Collection

Our approach and empirical study require data about Ethereum smart contracts and their associated transactional activities. We collected this data using Google BigQuery, a fully managed, cloud-native data warehouse offered by the Google Cloud Platform. Google maintains the `crypto_ethereum`[2] dataset using Ethereum ETL, a tool designed to extract, transform, and load data from the Ethereum blockchain into a structured format suitable for analysis (Google 2018; Ethereum ETL 2019).

[2] `bigquery-public-data.crypto_ethereum`

In addition, we used the Infura[3] service to interact with Ethereum smart contracts and retrieve on-chain data directly from the Ethereum blockchain. Infura provides developers with scalable and reliable access to the Ethereum network without requiring them to run their own node.

More specifically, we extracted data from the `crypto_ethereum` dataset, which contains information about transactions, contracts, blocks, events, tokens, and token holders. We focused on data from the `contracts` and `traces` tables. The `contracts` table contains metadata about deployed smart contracts (e.g., address, bytecode, creation timestamp), while the `traces` table captures granular details about smart contract executions (e.g., transaction hash, input, output, and execution result).

We used a snapshot of these tables spanning from August 7, 2015 to September 1, 2022, which marks the beginning of our experimental phase. As of that date, the `contracts` table contained records for 50,845,833 unique smart contracts, while the `traces` table included information on 1,695,517,186 distinct transactions and 5,503,071,306 individual traces.

To conduct our empirical evaluation, we used two distinct ground truth datasets. First, we compiled a dataset (GE) consisting of 3,177 contracts flagged as UPCs by Etherscan (Section 6.4), covering the period from January 2018 to September 2022. We used the GE dataset to evaluate Research Question 1 (Section 6.4). To further assess and benchmark the performance of UPC Sentinel against prior work, we utilized a second dataset (GB), released by Bodell III et al. (2023), which includes 994 labeled contracts, 825 UPCs and 169 Non-UPCs. This dataset was used to evaluate Research Questions 2 (Section 6.5), 3 (Section 6.6), and 4 (Section 6.7).

[3] https://www.infura.io/

In the following sections, we examine the characteristics of the GE and GB datasets and conduct systematic sanity checks to assess and validate their quality. Specifically, we follow established guidelines for evaluating data quality (Bhatia et al. 2024b), defining clear criteria for dataset modifications to ensure that each adjustment is both necessary and well-justified. Furthermore, we adhere to general empirical software engineering principles by thoroughly documenting every modification, along with its rationale and supporting evidence, to ensure transparency and reproducibility (Ralph et al. 2021). Lastly, we have made all results and findings available through our replication package, enabling peer verification and facilitating future research.

6.3.1 GE Ground Truth

Akin to prior studies (Bodell III et al. 2023), We carefully curated a ground truth dataset of UPCs to evaluate the accuracy of UPC Sentinel. Specifically, we employed our proxy detector method, described in Section 6.2.3, to extract all proxy contracts with at least two associated implementation contracts (Section 6.3). We then queried each identified proxy contract on Etherscan. If Etherscan reported at least two distinct implementation contracts for a proxy, we classified it as a UPC. This classification is grounded in the assumption that having two implementations typically signifies an upgrade (i.e., a transition from an earlier to a newer implementation contract).

Out of the 31,481 distinct filtered proxy contracts, Etherscan reported 3,177 (10.1%) as UPCs. We refer to this dataset as **GE**, which includes UPCs from January 2018 (marking the period when upgradeability proxy patterns began to be formally

recognized and widely adopted) through September 2022. Figure 6.12 illustrates the monthly distribution of collected UPCs. With the exception of four months (highlighted with red dots), at least one UPC was recorded in every month. The data reveals a steady rise in the number of UPCs from 2018 to 2022. Initially, the growth was modest during 2018 and 2019. However, from 2020 onward, there was a marked increase in the frequency of UPCs, indicating a growing adoption trend of upgradeability proxy patterns.

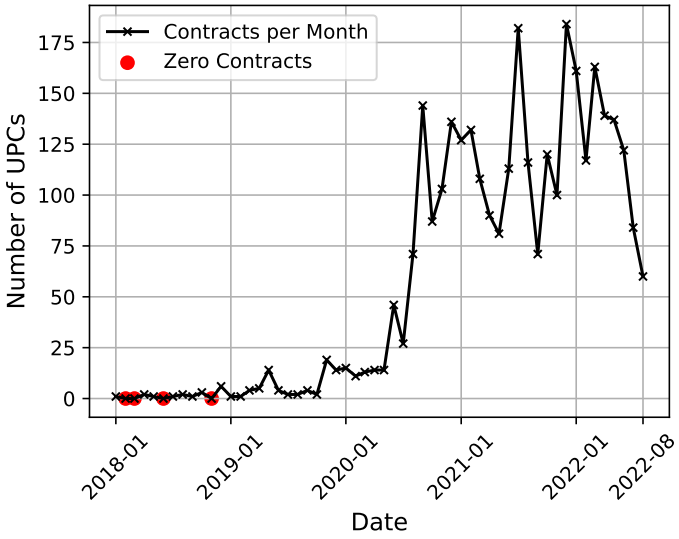


Figure 6.12: Monthly deployment trend of the 3,177 Upgradeable Proxy Contracts (UPCs) collected from Etherscan.

6.3.1.1 Mislabels

Mislabels refer to response variables that have been incorrectly assigned due to a flawed labeling strategy. This occurs when the label assigned to a data point does not reflect its true class or underlying characteristics, leading to systematic errors

that can compromise the accuracy and reliability of data-driven analyses and models (Bhatia et al. 2024b). Since the GE dataset may be influenced by the heuristics employed in Etherscan's Proxy Verification Service[4], we took additional steps to ensure the accuracy of labels by conducting a detailed review of the collected UPCs. However, given the labor-intensive nature of manually validating the entire dataset, we opted to examine a statistically representative random sample of contracts to strike a balance between accuracy and efficiency. The sample size ($n = 94$) was determined using Cochran's formula (Nanjundeswaraswamy and Divakara 2021), targeting a 95% confidence level with a 10% confidence interval. These parameters are widely accepted in empirical software engineering research to ensure sample representativeness (Yang et al. 2022b; Chen et al. 2021e; Zhou et al. 2021; Zhou et al. 2020a; Lin et al. 2019b; Lin et al. 2019a; Hassan et al. 2017; Hassan 2018).

For each contract, its associated metadata, including its address, source code, implementation contract addresses, and decompiled bytecode, was systematically reviewed. The first and third authors, each with significant experience in smart contract research (four years and six years, respectively), independently classified the 94 samples into UPC and Non-UPC categories. Leveraging established knowledge of UPC reference designs and proxy patterns, the authors (1) meticulously reviewed the source code of each contract, (2) identified proxy functionality, (3) located and analyzed the implementation variable, and (4) evaluated whether an upgrade function was implemented to modify the value of the implementation variable. If both authors independently agreed that a sample contract was a UPC or Non-UPC, the contract was labeled accordingly.

[4] https://etherscan.io/proxyContractChecker

In cases where the authors disagreed (two cases), they engaged in discussions to reach a consensus. Disagreements regarding the final label arose from two scenarios: (i) the inability to detect proxy functionality and determine where the proxy is implemented, and (ii) the inability to identify an upgrade function, particularly when it is implemented outside the proxy contract, either in an external contract or within the implementation contract itself. Despite these differences, the two authors agreed on the classification in 92 out of 94 cases, resulting in an observed proportionate agreement of **97.98%**. After resolving the disagreements, our manual verification confirmed that all reviewed samples were indeed UPCs. This outcome reinforces our confidence in the accuracy of the collected GE ground truth dataset. For additional details about this analysis, please refer to our replication package[5]. In the following, we explain the disagreement cases in more detail.

Case #1: Consider the case of `ZeroEx`[6]. The proxy functionality in `ZeroEx` is implemented within its `fallback` function. Its storage layout is defined in an external library called `LibProxyStorage`[7], which maintains a mapping between function selectors and the addresses of their corresponding implementation contracts. This structure provides developers with greater flexibility, allowing them to distribute roles and responsibilities (i.e., different services) across multiple implementation contracts that are registered within the mapping for client access.

When a `ZeroEx` client requests a service, the contract first retrieves the relevant implementation address from the mapping, then relays the call to that address. Our proxy detector identified 35 distinct implementation contracts to which `ZeroEx`

[5] `GE Quality Check.xlsx`
[6] https://etherscan.io/address/0xdef1c0ded9bec7f1a1670819833240f027b25eff#code#F1#L1
[7] https://etherscan.io/address/0xdef1c0ded9bec7f1a1670819833240f027b25eff#code#F8#L1

has delegated calls over time. Below is a summary of the discussion between the two authors (i.e., the first and third authors of this manuscript) who conducted a manual analysis of this proxy.

- Both authors confirmed that the `ZeroEx` proxy does not expose a public or external function allowing the owner to perform upgrades, indicating it does not follow the SMUP reference design. They also agreed that the proxy retrieves the implementation address from its own storage rather than from an external contract, meaning it does not conform to the ESUP reference design. As a result, they began examining `ZeroEx`'s implementation contracts to determine whether the upgrade function is implemented there, assessing whether `ZeroEx` follows the DUP reference design.

- The third author identified the upgrade function, while the first author did not, primarily due to the complexity and high number (35) of `ZeroEx` implementation contracts. The third author observed that only one of the implementation contracts, `SimpleFunctionRegistry`[8], is responsible for release management operations, such as adding, removing, or rolling back a service within the proxy's mapping, thus enabling contract evolution for other implementation contracts in `ZeroEx`. He also noted that `SimpleFunctionRegistry` is initialized during the `ZeroEx` setup phase, allowing release management capabilities from the outset. The first author, by contrast, expected each implementation contract to include an upgrade function similar to those outlined in

[8] https://etherscan.io/address/0x843c1ae003a7217a122e9e2a7336377ee51b660f/advanced#code#F1#L1

ERC-1822 Universal UPC Pattern. Without such direct upgrade function in every implementation contract, he found it difficult to follow the third author's reasoning and initially labeled the proxy as Non-UPC.

- After discussing their differing interpretations, both authors agreed that `ZeroEx` is a UPC. This case underscores the need for a detection method that analyzes all implementation contracts linked to a proxy, particularly for UPCs following the DUP reference design. As shown in Step 4 of Figure 6.7, UPC Sentinel addresses this by scanning all implementation contracts associated with a proxy to identify such complex UPCs.

Case #2: Consider the case of `BeaconProxy`[9]. The proxy functionality of `Beacon Proxy` is implemented within its `fallback` function, which first retrieves the address of the implementation contract and then relays the call to that address. To obtain the implementation address, `BeaconProxy` invokes the `childImplementation` getter function from an `IBeacon` contract, the address of which is stored in the proxy's storage. Below is a summary of the discussion between the two authors (i.e., the first and third authors of this manuscript) who conducted a manual analysis of this proxy.

- Both authors confirmed their understanding of the `BeaconProxy` and agreed that the implementation contract address is managed by an external contract, not by the proxy itself, indicating that `BeaconProxy` does not follow the SMUP or DUP reference designs. After examining the proxy's transaction history and storage on Etherscan, they identified that it calls an external contract, namely

[9] https://etherscan.io/address/0x87931e7ad81914e7898d07c68f145fc0a553d8fb#code

`AdminUpgradeabilityProxy`[10], to fetch the implementation contract address in advance of the `delegatecall`.

- At this point, the first author was able to identify the upgradeability functionality in `AdminUpgradeabilityProxy`, whereas the third author was not. Initially, the first author searched the source code of `AdminUpgradeability Proxy` to locate the `childImplementation` function but could not find it. Upon further investigation, he discovered that `AdminUpgradeabilityProxy` is itself a proxy that delegates to another implementation contract, `NFTXVault FactoryUpgradeable`[11]. By examining the source code of `NFTXVaultFactoryUpgradeable`, he located both the `childImplementation`[12] function and the `upgradeChildTo`[13] function, which handles upgrades. In contrast, the third author, while following a similar procedure, overlooked the fact that `AdminUpgradeabilityProxy` is itself a proxy and halted the analysis after failing to find `childImplementation` in its source code.

- After discussion, both authors agreed that `BeaconProxy` is a UPC. This example illustrates the second variant of the ESUP reference design, where the external contract (i.e., the target dependency) is a proxy. In such cases, it is necessary to analyze the implementation contracts of the target dependency to determine whether it qualifies as a UPC. As shown in Step B10 of Figure 6.6, UPC Sentinel handles this scenario by recursively examining the implementation contracts of the target dependency if it is identified as a proxy.

[10] https://etherscan.io/address/0xbe86f647b167567525ccaafcd6f881f1ee558216#code
[11] https://etherscan.io/address/0xfa750439a60e385d9e5f3d25eb9db289b74b7062#code
[12] https://etherscan.io/address/0xfa750439a60e385d9e5f3d25eb9db289b74b7062#readContract#F2
[13] https://etherscan.io/address/0xfa750439a60e385d9e5f3d25eb9db289b74b7062#writeContract#F17

6.3.2 GB Ground Truth

As part of their replication package, Bodell III et al. (2023) released a ground truth dataset (GB) used in their experiments. Since this dataset forms the basis for our evaluations and comparisons in Sections 6.5, 6.6, and 6.7, we begin by assessing the GB dataset using systematic guidelines for data quality analysis (Bhatia et al. 2024b). This step ensures that our evaluations are conducted with precision, reliability, and fairness. According to Bodell III et al. (2023), the GB ground truth dataset comprises 994 proxy contracts, categorized into 826 UPCs and 168 Non-UPCs. In the following subsections, we provide a detailed discussion of each quality criterion investigated.

6.3.2.1 Missing Values

Missing values refer to instances where certain data points are absent or not recorded in a dataset (Bhatia et al. 2024b). As part of their data collection, Bodell III et al. (2023) manually added 25 contracts to their ground truth dataset that notably lack associated Ethereum addresses, unlike the other entries. The absence of Ethereum addresses makes it impossible to verify whether these 25 contracts were sampled from live deployments or were artificially created. Moreover, UPC Sentinel requires a valid Ethereum address to retrieve essential contract-level information, such as bytecode, transaction history, and the full set of implementation contracts. As a result, we excluded these 25 instances from our analysis. After this exclusion, the remaining ground truth dataset contains 969 instances. For transparency, we flagged these 25 cases in our replication package, which is publicly available online[14].

[14] Missing Values.xlsx

6.3.2.2 Schema Violations

Schema violations occur when data does not conform to the expected structure or constraints defined by the schema (Bhatia et al. 2024b). Bodell III et al. (2023) reported that their ground truth dataset includes 994 labeled proxy contracts, implying that all labeled instances should conform to the "proxy" category. However, our review uncovered eight instances labeled as "NOT EVEN A PROXY!". In addition, these eight instances lacked predicted labels from USCHUNT, suggesting further inconsistencies. As a result, we excluded these cases from our analysis. After this step, the remaining ground truth dataset contains 961 instances. For transparency, we have flagged these cases in our replication package, which is publicly available online[15].

6.3.2.3 Duplicates

A duplicate refers to repeated entries in a dataset where one or more data points are identical to others, potentially introducing bias (Bhatia et al. 2024b). As discussed in Section 6.2.1, the input to UPC Sentinel is simply the address of the contract under examination. Since each contract address on Ethereum is globally unique and immutable, it serves as a distinct identifier for a specific deployed smart contract. However, we identified 40 clusters of duplicate entries in the GB dataset. Each cluster contained two duplicate entries, except for one cluster which contained three. To eliminate duplicate bias in our findings, we retained only one instance per cluster and removed the remaining duplicates. In 32 of the 40 clusters, the ground

[15] Schema Violations.xlsx

truth labels were consistent across all duplicates, so we randomly selected one instance from each cluster. In the remaining 8 clusters, we retained the instance whose ground truth label was reported as "y" (i.e., UPC), to avoid underestimating USCHUNT's performance in subsequent analyses. After removing duplicates, the resulting ground truth dataset contains 920 instances. For transparency, we flagged these 41 duplicate entries in our replication package, which is publicly available online[16].

6.3.2.4 Mislabels

Mislabels refer to response variables that have been incorrectly assigned due to a flawed labeling strategy (Bhatia et al. 2024b). This occurs when the assigned labels do not align with the true underlying data or the intended classification criteria, leading to systematic errors that can compromise the accuracy and reliability of data-driven analyses and models. Two commonly used approaches for identifying and addressing mislabels include: i) *Human Verification*, in which expert reviewers or third-party services manually validate and correct erroneous labels; and ii) *Programmatic Labeling*, which uses algorithms or heuristics to systematically detect and relabel inconsistent or incorrect data points (Bhatia et al. 2024b). In the following two subsections, we conduct sanity checks to evaluate the accuracy of the GB ground truth dataset, first in terms of correct *Proxy* labeling, and second in terms of correct *UPC* labeling.

6.3.2.4.1 A) Verification of Proxy Labels. In the first sanity checks, we systematically verify that the 920 contracts in the GB ground truth are, at a minimum,

[16] Duplicates.xlsx

legitimate proxy contracts. This verification combines multiple automated techniques, designed to be both effective and efficient, with manual validation to further increase confidence in the results.

As discussed in Section 6.2.3, proxy contracts are characterized by two inherent properties (Property #1 and Property #2), as identified in prior studies (Section 2.15.4). In Section 6.2.4.2, we described how we mapped these properties to the decompiled bytecode level through an automated analysis of a statistically representative random sample of proxy contracts. These properties were present in 99.98% of the sampled contracts; the remaining 0.02% represented rare edge cases. We leveraged these findings to perform a systematic analysis of the GB ground truth at the decompiled bytecode level. To ensure that these rare edge cases were not erroneously excluded, the automated detection was followed by manual verification.

Below, we detail the two verification steps involved in this process. For each step, we report the *observed proportion of agreement* (McHugh 2012) between our assigned labels and the original ground truth. The rationale for not using Cohen's Kappa in this specific context is discussed in the threats to validity section (Section 6.9.1).

- **A.1) Verifying Against Property #1.** This check focuses on verifying Property #1, which concerns the presence of a `delegatecall` statement. Figure 6.13 illustrates the protocol we followed to systematically conduct this verification. To mitigate potential issues related to decompilation errors or inaccuracies, we employed two widely used online decompilers, Panoramix and Dedaub, to assess whether the ground truth contracts contained a `delegatecall` statement in their decompiled bytecode (Step 1). Upon analysis, we found

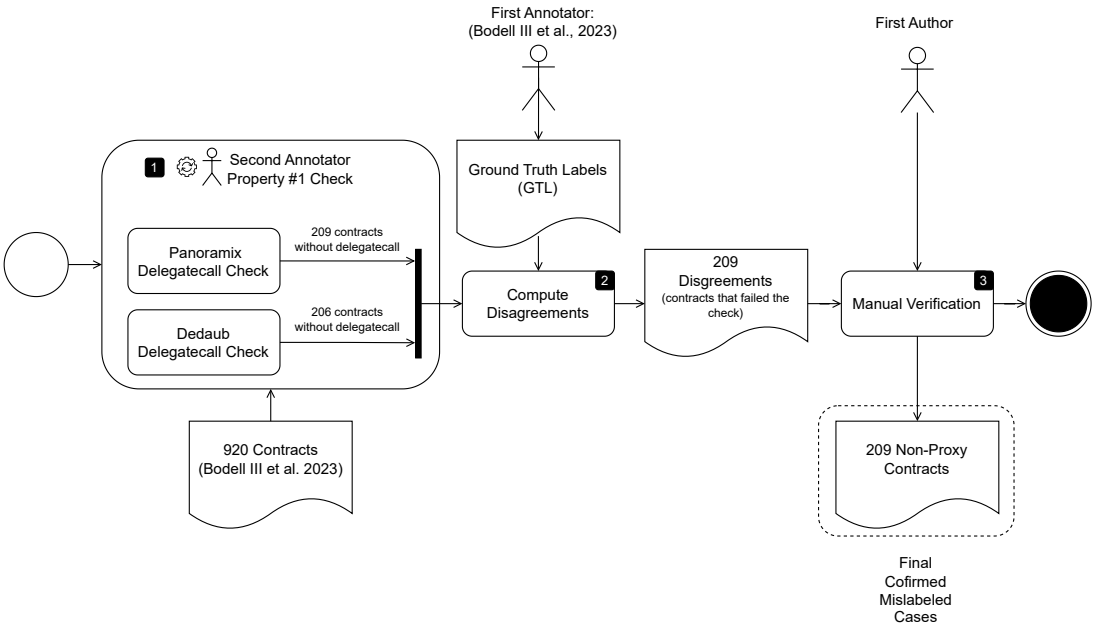


Figure 6.13: Protocol for examining the ground truth contracts with respect to Property #1 (i.e., the presence of `delegatecall`).

that in 206 contracts, both decompilers unanimously did *not* generate a `delegatecall` statement in the decompiled output. Additionally, in 3 contracts, Dedaub produced a `delegatecall` statement that was absent in Panoramix's output. However, upon further inspection, we observed that the `delegatecall` instructions generated by Dedaub were associated with library calls. Since Solidity internally uses `delegatecall` for invoking library functions, these do not indicate proxy behavior. Therefore, in these cases, Panoramix's output, which did not detect `delegatecall`, remains accurate with respect to identifying true proxy functionality.

This observation suggested that 209 contracts do not satisfy Property #1, indicating that they cannot, in practice, function as proxy contracts (Step 2).

In 711 out of the 920 analyzed contracts (77.3%), our automated check (i.e., the second annotator) aligned with the ground truth labels (i.e., the first annotator), confirming that these contracts satisfy Property #1 and could potentially be proxy contracts. This unexpected finding prompted further investigation. To resolve the discrepancies between our automatically verified labels and the original ground truth, and to better understand the underlying causes, the first author of this study, an expert with four years of experience in smart contract research, conducted a detailed manual analysis of the source code and UML class diagrams for these 209 contracts (Step 3). Both artifacts were retrieved from Etherscan. In all 209 cases, the first author agreed with the outputs of the two decompilers and was able to identify a consistent rationale explaining the absence of the `delegatecall` statement in each instance.

This analysis reveals that the verified source code of contracts retrieved from Etherscan can be noisy, often containing unreachable segments of code, particularly unreachable proxy functionality, in all 209 studied cases. Since the proxy functionality is not reachable from the main execution flow of the contract, the Solidity compiler does not generate any corresponding bytecode for it. This observation explains why none of the employed decompilers produced a `delegatecall` statement in the reconstructed source code. In Section 6.8.1, we discuss three recurring patterns of unreachable proxy functions observed in the dataset, along with specific examples. Further details on the remaining cases can be found in our replication package, which is publicly available online[17].

[17] A.1) Verify Against Property #1.xlsx

- **A.2) Verifying Against Property #2.** This check focuses on verifying Property #2, which concerns whether the proxy contract preserves the interface of the object it represents. Figure 6.14 illustrates the protocol we followed to systematically perform this verification process. Note that we excluded the 209 non-proxy contracts identified in the first sanity check, as they failed to satisfy Property #1 and therefore do not qualify for this analysis.

  As discussed in Section 6.2.4.2, Property #2 is satisfied when the `delegatecall` uses the first four bytes of the `calldata` (e.g., `call.data[return_data.size len 4]`, `call.data[0 len 4]`), or when the proxy and implementation function selectors are hard-coded and identical. We automatically analyzed the decompiled bytecode of the remaining 711 contracts (i.e., 920 total minus the 209 that failed Property #1) in Step 1. Our analysis found that 698 out of these 711 contracts (98.2%) satisfy Property #2 and are thus confirmed to be proxy contracts.

  To resolve the 13 disagreements between our automatically verified labels and the ground truth labels, the first author of this study conducted an additional manual review, examining the verified source code and UML class diagrams for each of the 13 identified instances (Step 3). Through this review, we found that in four cases, our automatic sanity check for Property #2 was incorrect, and the property was, in fact, satisfied, meaning the ground truth labels were accurate. These four missed instances correspond to `LendingPool`[18] contracts, where the proxy functionality is implemented within the `liquidationCall`[19] function. While the proxy functionality can be visually

[18] https://etherscan.io/address/0x2b6eaf43499b3db50df84b22c9451fe04599317c#code
[19] https://etherscan.io/address/0x2b6eaf43499b3db50df84b22c9451fe04599317c#writeContract#F5

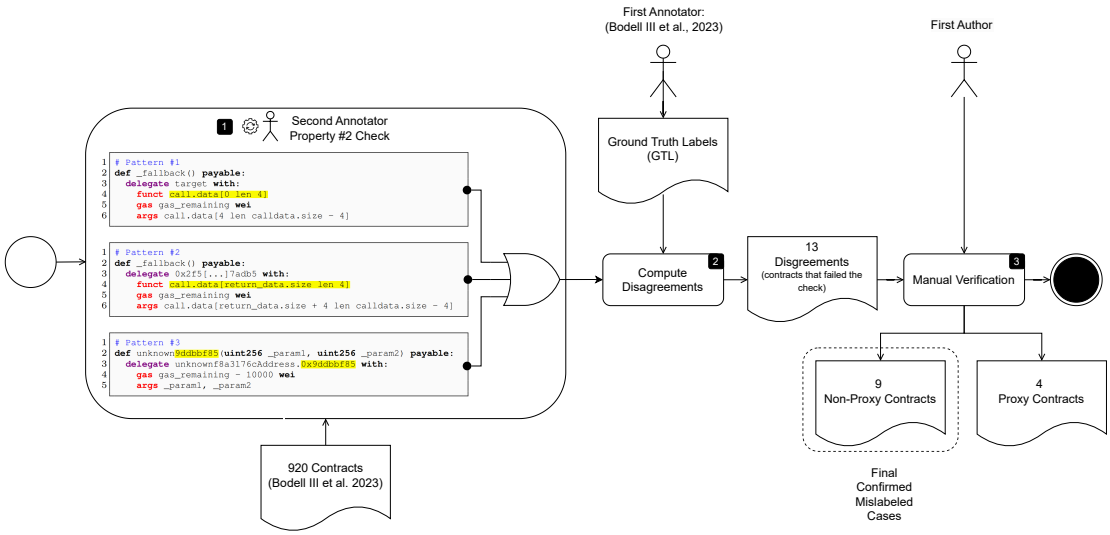


Figure 6.14: The protocol for examining the ground truth contracts against Property #2 (proxy must maintain the interface of the object it represents). The three patterns illustrating this property are highlighted in yellow.

confirmed at the source code level, detecting this specific case at the decompiled bytecode level requires a more sophisticated technique than the three lexical patterns currently used in our automatic analysis.

In the remaining nine cases, Property #2 was indeed not satisfied. Of these, eight involved `delegatecall` instructions related to library calls rather than proxy functionality, and one case involved a `delegatecall` used to implement an adapter contract rather than a proxy. An illustrative example involving a library-based `delegatecall` is the `EthVaultImpl`[20] contract. `EthVaultImpl` inherits from `EthVault`, which defines a `fallback` function implementing proxy functionality. However, `EthVaultImpl`'s own `fallback` function completely overrides that of its parent, effectively nullifying the inherited proxy

[20] https://etherscan.io/address/0xa6f8b3ce684e37144745daccb3ef3596e8ee745a#code

behavior. Consequently, `EthVaultImpl` does not function as a proxy. The observed `delegatecall` in the bytecode was related to one of its internal methods, specifically the `withdraw`[21] function, which invokes the `LibTokenManager` library. In addition, Listing 6.7 highlights the `SiuCoin`[22] contract, which serves as an adapter rather than a proxy. An adapter modifies the interface of the object it represents to ensure compatibility with the client. This intention is explicitly stated by the developers in lines 2 to 5 of the source code.

> *"This code allows us to redirect pre-signed calls with different function selectors to our own."*

Finally, line 9 demonstrates how the original function selector (`msg.sig`, i.e., the first four bytes of the calldata) is translated into a different selector before delegation. This behavior confirms that the contract modifies the interface rather than preserving it, violating Property #2. The full details of these nine cases are documented in our replication package, which is publicly available online[23].

After completing the two verification steps (A.1 and A.2), we identified that a total of 218 out of the 920 studied cases were not, in fact, proxy contracts as originally labeled in the ground truth. Consequently, we updated the ground truth labels for these 218 cases to Non-Proxy, resulting in a refined dataset consisting of 702 Proxy contracts and 218 Non-Proxy contracts.

[21] https://etherscan.io/address/0xa6f8b3ce684e37144745daccb3ef3596e8ee745a#writeContract#F16
[22] https://etherscan.io/address/0x1ff93b2d47d974644bddc62bf66940ebf18703e8#code
[23] `A.2) Verify Against Property #2.xlsx`

```
[...]
/**
 * @dev This code allows us to redirect pre-signed calls with different
 * function selectors to our own.
**/
function () public
{
   bytes memory calldata = msg.data;
   bytes4 new_selector = standardSigs[msg.sig];
   require(new_selector != 0);
   assembly {
     mstore(add(0x20, calldata), new_selector)
   }
   require(address(this).delegatecall(calldata));
   [...]
}
[...]
```

Listing 6.7: SiuCoin's `fallback` function implementing an adapter contract mechanism.

6.3.2.4.2 B) Verification of UPC Labels. In the previous sanity check (A), we identified 218 misclassified contracts that were originally labeled as proxy contracts by (Bodell III et al. 2023). However, after thorough verification, these contracts were determined not to be proxy contracts. Since being a proxy is a prerequisite for being classified as an upgradeability proxy, none of these 218 instances could be valid UPCs. Therefore, we reclassified these instances as Non-UPC. Among these 218 contracts, 157 were originally labeled as UP, and 61 were labeled as Non-UPC by (Bodell III et al. 2023). Accordingly, we switched the labels of the 157 misclassified UPC instances to Non-UPC, while the 61 correctly labeled instances remained unchanged. After this adjustment, the refined ground truth dataset consists of 621 UP and 299 Non-UPC contracts.

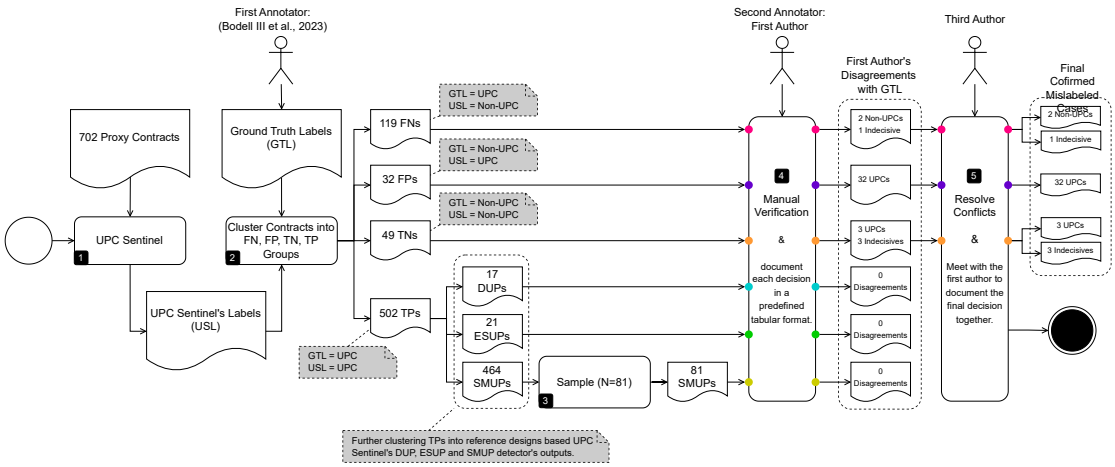


Figure 6.15: Protocol for examining the ground truth contracts against Property #2 (i.e., the proxy must maintain the interface of the object it represents). The three patterns illustrating this property are highlighted in yellow.

Given that we have confidently identified 218 contracts as Non-UPCs, we now focus on manually verifying the upgradeability labels of the remaining 702 contracts (i.e., $920-218$), following the protocol illustrated in Figure 6.15. Since verifying the ground truth labels for all 702 contracts is time-consuming and resource-intensive, we adopted a targeted strategy. Specifically, we applied UPC Sentinel to these contracts (Step 1) and clustered the results into four groups based on the outcomes: False Positives (FPs), False Negatives (FNs), True Negatives (TNs), and True Positives (TPs). This clustering enabled a focused and efficient validation process. Upon applying UPC Sentinel (Step 2), we observed the following distribution across the 702 contracts: 119 FNs, 32 FPs, 49 TNs, and 502 TPs.

For the FN, FP, and TN groups, we analyzed all instances. However, due to the resource-intensive nature of verifying the entire set of 502 TP instances, we opted to verify a representative sample from the TP group. To mitigate potential sampling bias, we first stratified the TP instances based on the reference design category,

DUP, ESUP, or SMUP, as identified by UPC Sentinel. Among the 502 TP cases, 17, 21, and 464 instances were classified as adhering to the DUP, ESUP, and SMUP reference designs, respectively. Subsequently, we verified the ground truth labels for all 17 DUP and 21 ESUP instances, along with a statistically representative random sample of 81 SMUP instances (Step 3), yielding a total of 119 manually verified TP cases. The sample size for the SMUP group was determined using Cochran's formula (Nanjundeswaraswamy and Divakara 2021), targeting a 95% confidence level and a 10% confidence interval. These parameters are commonly adopted in empirical software engineering research to ensure robust and representative sampling (Hassan 2018; Lin et al. 2019a; Lin et al. 2019b; Zhou et al. 2020b; Zhou et al. 2020a; Chen et al. 2021d; Yang et al. 2022a).

For each contract, its associated metadata, including source code, implementation contract addresses, transaction history, and decompiled bytecode, served as input to the verification process. Guided by the upgradeability reference designs and patterns (Section 6.2.4.1 and Appendix 2.15.6), the first author manually analyzed all 119 FNs, 32 FPs, 49 TNs, and the 119 sampled TPs to determine whether each contract is a UPC or Non-UPC. Since some instances were inactive, the first author labeled a contract as *indecisive* (Step 4) when a definitive classification could not be made due to missing or incomplete information (e.g., uninitialized implementation contract or unresolved external addresses). If the label assigned by the first author (i.e., second annotator) differed from the original ground truth label established by Bodell III et al. (2023) (i.e., first annotator), which occurred in 41 instances, the third author conducted an independent review and made the final judgment on the label (Step 5).

In cases where we refined a ground truth label, both the first and third authors collaboratively documented the findings in our replication package. Specifically, for contracts confirmed as UPCs (35 out of 41 instances), we recorded the details in a systematic tabular format. This format includes key information such as the contract's name and the specific function in which proxy functionality is implemented, the implementation variable, the confirmed upgradeability reference design (i.e., SMUP, ESUP, or DUP), the specific upgradeability pattern, the contract in which the upgrade function resides, the line number where the upgrade function begins, and, when applicable to DUP or ESUP designs, the address of the implementation or external contract containing the upgrade function. This structured documentation ensures that our analyses are fully reproducible, enabling others to verify our results, replicate our methodology, and apply the same approach to identify upgrade mechanisms in newly discovered UPCs. For the indecisive cases (4 out of 41 instances), as well as the two cases where the ground truth label was UPC but our analysis determined them to be Non-UPC, we have provided a detailed rationale supporting our decisions.

The following presents the detailed analysis conducted for each group, accompanied by concrete examples. For each group, we report the observed percentage of agreement (McHugh 2012) between the labels assigned by the first author and those originally reported in the ground truth dataset. A discussion on why Cohen's Kappa was deemed unsuitable for this context is provided in the threats to validity section (Section 6.9.1).

- **B.1) Verifying False Negatives.** FNs are instances whose original ground truth label is UPC, but they are identified as Non-UPC by UPC Sentinel. Out of

all 119 FNs, in three instances the labels assigned by the first author deviated from the ground truth "UPC" label. Among these, two were labeled as Non-UPC and one as *indecisive* by the first author, with both classifications later confirmed by the third author. Thus, the observed proportion of agreement between the ground truth labels and the first author's assessment is 97.5%. The *indecisive* contract[24] is an inactive proxy whose implementation contract address[25] was never initialized by developers. Consequently, we could not fully verify its upgradeability status to determine whether the upgrade logic is implemented inside the implementation contract. We revised the labels for the two confirmed Non-UPC instances and excluded the one indecisive instance from the ground truth. Following this adjustment, the refined ground truth now comprises 618 UPC contracts and 301 Non-UPC contracts. Detailed documentation supporting the classification of these three cases is included in our replication package[26].

- **B.2) Verifying False Positives.** FPs are instances whose original ground truth label is Non-UPC but are identified as UPC by UPC Sentinel. Upon analysis, the first author determined that all 32 FP instances are indeed UPCs. Thus, the observed proportion of agreement between the ground truth labels and the first author's assessment is 0%. Following our verification protocol, these 32 cases were subsequently reviewed by the third author and confirmed to be UPCs. Consequently, we revised the labels for these 32 instances from Non-UPC to UPC.

[24] https://etherscan.io/address/0x239a9de61b2d6b1dc9695562a336343e819571e4#code
[25] https://etherscan.io/address/0x239a9de61b2d6b1dc9695562a336343e819571e4#readContract#F1
[26] B.1) Verifying False Negatives.xlsx

Following these adjustments, the refined ground truth now comprises 650 UPC contracts and 269 Non-UPC contracts. Detailed documentation supporting why these 32 cases are classified as UPCs is included in our replication package online[27]. Specifically, among these, 27 conform to the DUP upgradeability reference design, where the upgrade mechanism is implemented within the proxy's implementation contract, and five conform to the SMUP upgradeability reference design, where the upgrade mechanism resides directly within the proxy contract itself. For example, the `Proxy`[28] contract is an instance of the ERC-1822 Universal UPC Pattern, with its implementation contract, `TokenSoftTokenV2`[29], containing the `updateCodeAddress`[30] function, which facilitates upgrades in accordance with the ERC-1822 Universal UPC Pattern. As another example, the `NFTFactoryProxy`[31] contract follows the Unstructured Storage Upgradeability Proxy Pattern, with its upgrade function, `setLogicContract`[32], implemented directly within the proxy contract. Consequently, we have updated the ground truth by reclassifying these 32 instances from Non-UPC to UPC.

- **B.3) Verifying True Negatives.** TNs are instances whose original ground truth label is Non-UPC and are also identified as Non-UPC by UPC Sentinel. Among the 49 TNs, the first author's assigned labels deviated from the ground truth in six instances. Thus, the observed proportion of agreement between the ground truth label and the first author is 87.7%. Following our verification

[27] B.2) Verify False Positives.xlsx
[28] https://etherscan.io/address/0x5f6d994e6ba25a0a23efae15e46a6e79527bdf3f#code
[29] https://etherscan.io/address/0x7aac67b8cb7f39e080672ca6a32f5a6a964c26a4#code
[30] https://etherscan.io/address/0x7aac67b8cb7f39e080672ca6a32f5a6a964c26a4#writeContract#F34
[31] https://etherscan.io/address/0x8f9749054390b96cca7b5a41c7215cb6745daaee#code
[32] https://etherscan.io/address/0x8f9749054390b96cca7b5a41c7215cb6745daaee#writeContract#F1

protocol, the third author analyzed and resolved the disagreements. Out of these six cases, three were originally labeled as Non-UPC but were confirmed to be UPCs. For example, `BZxProxy`[33] is a SMUP-based UPC contract that includes an upgrade function named `setTarget`[34], which facilitates replacing its implementation contract. Another example is `DutchExchangeProxy`[35], a DUP-based UPC whose upgrade function, `startMasterCopyCountdown`[36], is implemented inside its corresponding implementation contract[37]. We refined the ground truth by reclassifying these three cases as UPCs. The remaining three instances were confirmed to be *indecisive*, for reasons similar to those discussed in Section 6.3.2.4.2. Following these adjustments, the refined ground truth now comprises 653 UPC contracts and 263 Non-UPC contracts. Further documentation for this analysis is provided in our replication package[38].

- **B.4) Verifying True Positives.** TPs are instances whose original ground truth label is UPC and are also identified as UPC by UPC Sentinel. Using our established verification protocol (Figure 6.15), the first author manually reviewed all 119 contracts to determine whether they indeed represent UPCs, as indicated by the ground truth. Upon analysis, the first author fully concurred with the ground truth labels, confirming all 119 instances as UPCs. This results in an observed agreement rate of 100%. Detailed documentation justifying the classification of these contracts is available in our online replication package[39].

[33] https://etherscan.io/address/0x86343be63c60ce182d8b5ac6a84f0722d8d61ae5#code
[34] https://etherscan.io/address/0x86343be63c60ce182d8b5ac6a84f0722d8d61ae5#writeContract#F8
[35] https://etherscan.io/address/0xe938e574afef9a75d8d6b6b5bd7dc44f3df1add9#code
[36] https://etherscan.io/address/0x902B4b1882621e02893A1330aa42221939B27447#writeContract#F21
[37] https://etherscan.io/address/0x902B4b1882621e02893A1330aa42221939B27447#code
[38] B.3) Verify True Negatives.xlsx
[39] B.4) Verify True Positives.xlsx

After completing the verification steps (B.6.3.2.4.2–6.3.2.4.2), the refined ground truth now comprises 653 confirmed UPCs and 263 confirmed Non-UPCs, totaling 916 contracts. Notably, as discussed in Sections 6.3.2.4.2 and 6.3.2.4.2, four cases were excluded due to insufficient evidence or missing information, which prevented us from definitively determining their upgradeability status.

6.4 RQ1: How Accurately Does UPC Sentinel Detect Upgradeable Proxy Contracts in the GE Dataset?

6.4.1 Motivation

In Section 6.3.1, we introduced the GE dataset, a novel, curated ground truth consisting of 3,177 confirmed Upgradeable Proxy Contracts (UPCs). This dataset was meticulously compiled through a combination of automated filtering and manual validation to ensure high confidence in the correctness of each labeled instance. Given its scale and accuracy, GE serves as a reliable benchmark for evaluating UPC detection methods. This research question is motivated by the need to assess how effectively UPC Sentinel can identify upgradeable proxy contracts when evaluated against a large, diverse, and verified dataset. By applying UPC Sentinel to GE, we aim to measure the tool's detection accuracy in a realistic setting and validate its practical utility in identifying real-world upgradeable proxies. The results provide foundational insights into the reliability and robustness of UPC Sentinel before comparing it to other state-of-the-art techniques in subsequent research questions.

6.4.2 Experimental Setup

We evaluate UPC Sentinel on the GE ground truth dataset (Section 6.3.1) and report its performance using the accuracy metric. In this context, accuracy is defined as the proportion of correctly identified Upgradeable Proxy Contracts (UPCs) among all instances in the dataset. Since the GE dataset exclusively comprises 3,177 smart contracts that are all confirmed UPCs, every instance represents a positive case. As a result, there are no negative cases in this setting, i.e., no False Positives or True Negatives. Therefore, consistent with the evaluation methodology of (Bodell III et al. 2023), accuracy is equivalent to the recall metric in this scenario.

6.4.3 Findings

Observation 6.1) ***UPC Sentinel exhibits an accuracy of 99% on the GE ground truth.*** Out of the total 3,177 UPCs, UPC Sentinel successfully detected 3,147 instances. However, it failed to identify 30 UPCs. An analysis of the intermediate outputs reveals that among the 3,147 detected cases, 2,540 (80.7%), 328 (10.4%), and 279 (8.9%) align with the SMUP, DUP, and ESUP reference designs, respectively. Additionally, 79 instances (2.5%) correspond to variant #2 of either the ESUP or DUP reference designs, where the upgrade functions reside in transitive dependencies (Section 6.2.4.1). A detailed example is provided in Section 6.3.1.1.

Error analysis. Among the 30 undetected UPCs, 22 were missed due to decompilation issues that prevented the recovery of key control-flow or storage layout information. For the remaining eight instances, UPC Sentinel was unable to classify them due to peculiar decompilation patterns.

6.5 RQ2: How Does UPC Sentinel's Proxy Detector Compare to USCHUNT in Detecting Proxy Contracts?

6.5.1 Motivation

Identifying whether a given smart contract is a proxy is a fundamental prerequisite for determining if it qualifies as an Upgradeable Proxy Contract (UPC) or a Forwarder Proxy Contract (FPC). In a typical UPC analysis workflow, the initial step involves classifying contracts as either proxy or non-proxy. Only those identified as proxy contracts undergo further analysis to determine whether they follow a UPC or FPC design. Consequently, the accuracy and reliability of proxy detection critically impact the effectiveness of the entire UPC detection pipeline.

This research question aims to evaluate the performance of UPC Sentinel's proxy detection module by comparing it to USCHUNT, a state-of-the-art method introduced by Bodell III et al. (2023). Specifically, we assess the extent to which each approach correctly distinguishes proxy contracts from non-proxies in the refined GB ground truth dataset (Section 6.3.2). A superior proxy detection capability ensures higher-quality downstream results, such as reduced False Positives or False Negatives in proxy type classification (Section 6.2.4). Therefore, this comparison provides critical insights into the foundational accuracy and practical applicability of the two techniques in real-world analysis of smart contract ecosystems.

6.5.2 Experimental Setup

For this case study, we utilize the refined GB ground truth dataset (Section 6.3.2), originally published by Bodell III et al. (2023), which spans contract data from July 2017 to November 2021. While the original dataset included 994 contracts labeled as proxies, we conducted a series of systematic quality control procedures to enhance its reliability. These procedures addressed missing values (Section 6.3.2.1), schema violations (Section 6.3.2.2), duplicate entries (Section 6.3.2.3), and labeling inconsistencies (Section 4.2.4.A)). As a result, the refined GB ground truth now contains 702 confirmed Proxy contracts and 218 confirmed Non-Proxy contracts. To evaluate proxy detection performance, we compare UPC Sentinel's proxy detector against USCHUNT (Bodell III et al. 2023) under two distinct settings:

i. **Mixed Setting**. This setting includes both active and inactive proxy contracts to comprehensively evaluate the performance of each approach. It offers insights into how UPC Sentinel and USCHUNT handle the entire range of proxy contracts, regardless of their operational state.

ii. **Active-Only Setting**. This setting considers only active proxy contracts, those whose proxy functionality has been invoked at least once after deployment, helping us measure UPC Sentinel's effectiveness under our defined scope (Section 6.2.3) where proxies are functioning normally as intended.

When examining all 702 proxy contracts to determine their activity status as of September 2022, we identified 122 instances as inactive. Table 6.2 presents the distribution of labels in the ground truth under each setting. To evaluate UPC Sentinel, we run its proxy detection module separately in both settings. We then compute the Precision, Recall, and F1-score metrics and compare the performance

Table 6.2: Comparison of label distribution across Mixed and Active-Only settings.

Setting	**Label**		**#Total**
	#Proxy	*#Non-Proxy*	
Mixed	702	218	920
Active-Only	580	218	798

of UPC Sentinel with that of USCHUNT. To assess whether the observed differences in performance metrics are statistically significant, we apply McNemar's test (with a significance threshold of $p = 0.05$), a non-parametric test suitable for analyzing paired nominal data, such as binary classification results (Mcnemar 1947). Finally, we perform an error analysis of False Positives and False Negatives to derive additional insights.

6.5.3 Findings

Observation 6.2) ***UPC Sentinel's proxy detector demonstrates statistically superior performance to USCHUNT in both settings, particularly in terms of precision and F1-score.*** Table 6.3 presents a comparison between the performance of UPC Sentinel and the USCHUNT method across both the Mixed and Active-Only settings. In the Mixed setting, UPC Sentinel achieved a perfect precision of 100% for the Proxy class, though its recall was slightly lower at 82.6%, resulting in an F1-score of 90.5%. In contrast, USCHUNT reached 100% recall for the Proxy class but with lower precision (76.3%), yielding an F1-score of 86.6%. For the Non-Proxy class, UPC Sentinel again showed strong performance, achieving 100% recall, though with reduced precision (64.1%) and an F1-score of 78.1%. USCHUNT, on

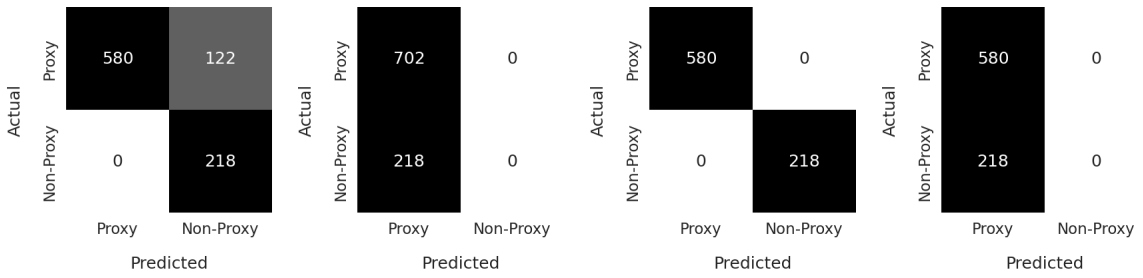
Table 6.3: Comparison of UPC Sentinel's proxy detector and USCHUNT in Mixed and Active-Only settings.

Setting	Method	Precision	Recall	F1	Precision	Recall	F1
		Proxy (702)			*Non-Proxy (218)*		
Mixed	UPC Sentinel	**100**	82.6	**90.5**	**64.1**	**100**	**78.1**
	USCHUNT	76.3	**100**	86.6	0	0	0
		Proxy (580)			*Non-Proxy (218)*		
Active-Only	UPC Sentinel	**100**	**100**	**100**	**100**	**100**	**100**
	USCHUNT	72.7	**100**	84.2	0	0	0

the other hand, produced no correct classifications for Non-Proxy data, resulting in 0% precision, recall, and F1-score.

In the Active-Only setting, where inactive proxy contracts are excluded, UPC Sentinel maintained perfect performance across both Proxy and Non-Proxy labels, achieving 100% precision, recall, and F1-score. In contrast, USCHUNT sustained a high recall of 100% for the Proxy class but exhibited lower precision at 72.8%, resulting in an F1-score of 84.2%. For the Non-Proxy class, USCHUNT again failed to identify any correct instances, yielding 0% precision, recall, and F1-score. McNemar's test confirms that the observed performance difference between UPC Sentinel and USCHUNT is statistically significant.

Error analysis. Figure 6.16 presents the confusion matrices for both UPC Sentinel and USCHUNT under the Mixed and Active-Only settings. In the Mixed setting, UPC Sentinel achieved a recall of 82.6% because it does not detect the 122 inactive proxy contracts (Section 6.5.2). As UPC Sentinel classifies inactive proxies as Non-Proxy contracts, this leads to a lower precision (64.1%) for the Non-Proxy class. Conversely, USCHUNT attained lower precision (76.3%) in the Proxy class due to the 218 misclassified instances whose labels were corrected during quality control (Section 4.2.4.A).



a. Mixed setting UPC Sentinel b. Mixed setting USCHUNT c. Active-Only setting UPC Sentinel d. Active-Only setting USCHUNT

Figure 6.16: Confusion matrices for proxy detection by UPC Sentinel and USCHUNT in Mixed and Active-Only settings.

In the Active-Only setting, UPC Sentinel achieved perfect scores for precision, recall, and F1 across both classes. However, USCHUNT's performance again suffered due to the same 218 misclassified instances, leading to reduced precision for the Proxy class. Since USCHUNT fails to correctly detect these Non-Proxy contracts, its precision, recall, and F1-score for the Non-Proxy class are reported as 0% in both settings.

6.6 RQ3: How Does UPC Sentinel's Proxy Type Classifier Compare to USCHUNT in Classifying Proxy Contracts?

6.6.1 Motivation

While identifying whether a contract is a proxy is a crucial first step, the core goal of UPC analysis is to determine whether a proxy contract is used for contract evolution or not. This is particularly important in the Ethereum ecosystem, where

upgradeable proxy contracts (UPCs) introduce critical considerations related to system mutability, governance, and potential security risks. In this research question, we focus on evaluating the ability of UPC Sentinel to correctly classify proxy contracts as either Upgradeable Proxy Contracts (UPCs) or Forwarder Proxy Contracts (FPCs), and compare its performance against USCHUNT Bodell III et al. (2023), a state-of-the-art static analysis tool operating at the source code level.

This comparison is particularly significant for two reasons. First, a misclassification at this stage can lead to incorrect assumptions about the contract's ability to evolve post-deployment, impacting downstream decisions such as vulnerability assessment, auditing, or governance design. Second, while USCHUNT depends on source code availability, UPC Sentinel operates at the bytecode level, making it applicable even to unverified contracts. Therefore, this evaluation provides insights into the practical robustness and applicability of both tools in real-world contract ecosystems.

6.6.2 Experimental Setup

For this experiment, we use the refined GB ground truth (Section 6.3.2). Following systematic quality control checks, including evaluations for missing values (Section 6.3.2.1), schema violations (Section 6.3.2.2), duplicates (Section 6.3.2.3), and mislabels (Sections 4.2.4.A and 4.2.4.B), the refined GB ground truth now consists of 653 UPC contracts and 263 Non-UPC contracts.

As in RQ2 (Section 6.5), we compare the performance of UPC Sentinel with USCHUNT in both the Mixed and Active-Only settings (Section 6.5.2). The comparison focuses on Precision, Recall, and F1-measure performance metrics for detecting

Table 6.4: Comparison of UPC and Non-UPC distributions across Mixed and Active-Only settings.

Setting	**Label**		**#Total**
	#UPC	*#Non-UPC*	
Mixed	653	263	916
Active-Only	538	260	798

upgradeable proxy contracts. Table 6.4 presents the label distribution for both the Mixed and Active-Only settings. Additionally, we apply McNemar's test (p-value = 0.05) Mcnemar (1947) to assess whether the observed differences in performance metrics are statistically significant. Finally, we conduct an error analysis of False Positives and False Negatives to gain further insights.

6.6.3 Findings

Observation 6.3) ***In the Mixed setting, UPC Sentinel excels in precision and overall F1-score, while USCHUNT offers higher recall for UPCs.*** Table 6.5 compares the performance of UPC Sentinel's upgradeability detector with the USCHUNT method in both Mixed and Active-Only settings. Under the Mixed setting, UPC Sentinel outperforms USCHUNT in terms of precision, achieving a perfect score of 100% for UPCs, compared to USCHUNT's 79.5%. This indicates that UPC Sentinel is highly accurate in detecting UPCs without generating False Positives.

However, with respect to recall, USCHUNT demonstrates stronger performance, achieving a recall of 94.2% versus UPC Sentinel's 81.8%. This suggests that USCHUNT identifies a greater proportion of UPCs, though at the cost of increased False Positives, as reflected in its lower precision. The F1-score, which balances precision

and recall, favors UPC Sentinel (90%) over USCHUNT (86.2%). Regarding the Non-UPC class, the two methods show notable differences. UPC Sentinel achieves a precision of 68.8% and a perfect recall of 100%, meaning it correctly identifies all Non-UPC contracts, albeit with some False Positives. In contrast, USCHUNT attains a higher precision of 73.2% but a substantially lower recall of 39.5%, resulting in a significantly lower F1-score of 51.4%. Finally, McNemar's test confirms that the performance difference between UPC Sentinel and USCHUNT in this setting is statistically significant.

Error analysis. Figure 6.17.a and Figure 6.17.b show the confusion matrices for UPC Sentinel and USCHUNT, respectively, under the Mixed setting. UPC Sentinel failed to detect 119 UPCs, of which 96.6% (115 instances) were inactive (i.e., outside the scope of our approach), and only 3.4% (4 instances) were active proxy contracts. These inactive cases could not be detected because UPC Sentinel relies on transaction data from proxy contracts to function effectively (Section 6.2.3). A detailed analysis of the four misclassified active instances revealed that three failures were caused by the static analyzer's inability to identify a relevant `delegatecall` *statement (Section 6.2.4), while the remaining case was attributed to an unconventional contract storage design. In contrast, USCHUNT misclassified 159 Non-UPCs as UPCs. These 159 instances correspond to the mislabels we corrected in Section 4.2.4.B. Additionally, USCHUNT failed to detect 38 actual UPCs, of which 35 were initially mislabeled in the ground truth and were later corrected from Non-UPC to UPC (also discussed in Section 4.2.4.B). The remaining 3 instances were genuinely missed by USCHUNT.*

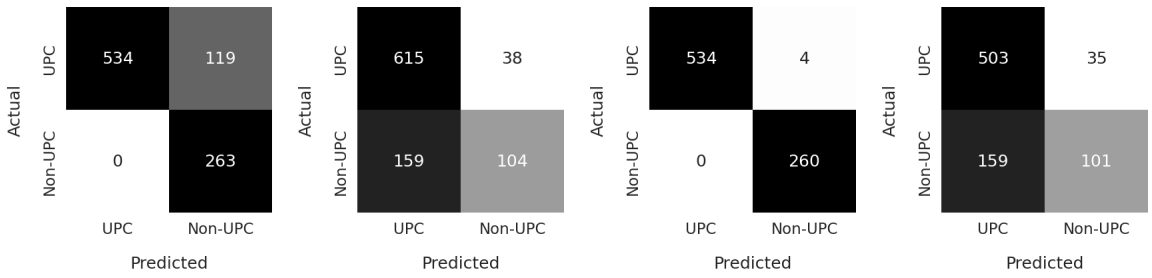
Observation 6.4) ***In the Active-Only setting, UPC Sentinel demonstrates superior performance compared to USCHUNT across both UPC and Non-UPC categories.***

Table 6.5: Comparison of UPC Sentinel's proxy type classifier and USCHUNT in Mixed and Active-Only settings.

Dataset	Method	Precision	Recall	F1	Precision	Recall	F1
		UPC (653)			*Non-UPC (263)*		
Mixed	UPC Sentinel	**100**	81.8	**90**	68.8	**100**	**81.5**
	USCHUNT	79.5	**94.2**	86.2	**73.2**	39.5	51.4
		UPC (538)			*Non-UPC (260)*		
Active-Only	UPC Sentinel	**100**	**99.3**	**99.6**	**98.5**	**100**	**99.2**
	USCHUNT	76	93.5	83.8	74.3	38.9	51

Specifically, UPC Sentinel achieves 100% precision and 99.3% recall for UPCs, resulting in an F1 score of 99.6%. In contrast, USCHUNT reports a precision of 76% and a recall of 93.5% for UPCs, yielding a lower F1 score of 83.8%. For the Non-UPC category, UPC Sentinel again excels with a precision of 98.5% and a perfect recall of 100%, resulting in an F1 score of 99.2%. Meanwhile, USCHUNT struggles in this category, achieving only 74.3% precision and a significantly lower recall of 38.9%, which leads to an F1 score of just 51%. Finally, McNemar's test confirms that the performance differences between UPC Sentinel and USCHUNT in this setting are statistically significant.

Error analysis. Figures 6.17.c and 6.17.d present the confusion matrices for UPC Sentinel and USCHUNT, respectively, under the Active-Only setting. In this setting, which focuses exclusively on active proxy contracts, UPC Sentinel failed to detect only 4 UPCs, as noted in Observation 3. In contrast, USCHUNT misclassified 159 Non-UPC contracts as UPCs and incorrectly labeled 35 UPCs as Non-UPCs. These 194 cases correspond to instances whose upgradeability labels were refined during the ground truth validation process described in Section 4.2.4.B.



a. Mixed setting UPC Sentinel b. Mixed setting USCHUNT c. Active-Only setting UPC Sentinel d. Active-Only setting USCHUNT

Figure 6.17: Confusion matrices for proxy type classification by UPC Sentinel and USCHUNT in Mixed and Active-Only settings.

6.7 RQ4: How Does UPC Sentinel's UPC Pattern Classifier Compare to USCHUNT in Classifying UPC Patterns?

6.7.1 Motivation

RQ3 focused on determining whether UPC Sentinel can detect the presence of mechanisms that enable contract evolution (i.e., distinguishing UPCs from FPCs). Building on this, RQ4 examines UPC Sentinel's ability to identify the specific UPC pattern that each UPC implements.

Accurate identification of these patterns is essential for stakeholders involved in contract development, auditing, and research. Each design introduces distinct operational behaviors, varying levels of flexibility, and unique security risks. For example, some designs centralize upgrade authority in a single privileged address,

while others distribute upgrade responsibilities across modular components. Understanding the precise pattern in use enables more targeted security assessments and enhances the traceability and transparency of contract changes over time.

This research question empirically evaluates the performance of UPC Sentinel's Pattern Classifier layer (Section 6.2.5), which assigns each detected UPC to a known UPC pattern. Additionally, this evaluation compares UPC Sentinel's bytecode-level classification with USCHUNT's source-level detection, highlighting the finer-grained and pattern-aware capabilities of UPC Sentinel.

6.7.2 Experimental Setup

Given the goal of this research question, evaluating the performance of the upgradeability pattern classifier layer, we include all 538 active UPCs from the GB ground truth in our analysis (Table 6.4). For each UPC entry, the GB ground truth specifies the corresponding upgradeability pattern(s); in some cases, a single contract is associated with multiple patterns. In total, the 538 UPCs are categorized into 19 groups based on the combination of patterns implemented by each UPC. The second column of Table 6.6, namely "#Active UPCs", displays the distribution of active UPCs across these groups. For instance, the Transparent+ERC-1967 category, representing the largest group (43.7%), consists of UPCs that employ the ERC-1967 storage slot in combination with the Transparent UPC Pattern.

Note that a small subset of 11 data points combines ERC-1967, Unstructured, or Inherited Storage layout principles with patterns from other reference designs, such as ERC-1822, Registry, or Beacon. These combinations reflect design considerations aimed at mitigating storage collisions, as discussed in Section 2.15.2. Although

such cross-design pattern combinations may appear counterintuitive, Bodell III et al. (2023) provided no further details or rationale for these implementations. Given their rarity and negligible influence on the overall analysis, we retained the ground truth labels as originally provided.

In addition, the replication package includes a column indicating whether USCHUNT successfully identified the specific pattern of each UPC, which we used to compute its performance. This column contains three possible values: “y”, “n”, or “just says it is upgradable”. The value “y” denotes cases where USCHUNT correctly identified the upgradeability pattern; “n” indicates cases where it missed both the UPC and its corresponding pattern; and “just says it is upgradable” refers to instances where USCHUNT detected that the contract is upgradeable but did not specify a known pattern, either because the contract did not conform to any predefined patterns or because USCHUNT failed to identify it. As shown in Table 6.6, the ground truth includes a “Non-pattern” category. For this category, we consider USCHUNT’s prediction a match if its label is either “y” or “just says it is upgradable”, to avoid underestimating its performance. For all other patterns, a match is only counted when the label is explicitly “y”.

In Section 4.2.4.B (sanity checks on correct upgradeability labeling), we systematically identified 35 mislabeled instances that were reclassified from Non-UPC to UPC (see Steps 4.2.4.B.2 and 4.2.4.B.3 for details). This set includes 32 active and 3 inactive contracts. Since our evaluation in this research question focuses exclusively on active cases, only the 32 active instances are treated as false negatives (i.e., labeled “n”) for USCHUNT.

As stated in our methodology, since our approach always identifies the most specific upgradeability pattern (Section 6.2.5) within the reference design a UPC adheres to, we consider it a positive match if the pattern identified by our method is a subset of the ground truth label. Similarly, for the 11 cases where the ground truth label combines patterns across reference designs, we treat UPC Sentinel's output as a positive match if the identified pattern is a subset of the contract's ground truth label.

Finally, similar to RQ1 (Section 6.4), we report the performance for each group using the accuracy metric, which highlights the percentage of patterns successfully identified by each method. Additionally, we perform McNemar's exact test (suitable for small sample sizes) at a significance level of $p = 0.05$ within each group to evaluate whether the observed differences in performance metrics between UPC Sentinel and USCHUNT are statistically significant.

6.7.3 Findings

Observation 6.5) ***UPC Sentinel and USCHUNT perform comparably in terms of pattern detection, with each method outperforming the other in one pattern group.*** Table 6.6 presents the accuracy of UPC Sentinel and USCHUNT across 19 groups of patterns. Specifically, USCHUNT outperforms UPC Sentinel in the Unstructured pattern group, achieving an accuracy of 98.55% compared to 85.51%. Conversely, UPC Sentinel outperforms USCHUNT in the *Non-pattern* group, attaining an accuracy of 85.19% compared to 55.56%. According to McNemar's test, these differences are statistically significant.

Table 6.6: Accuracy comparison between UPC Sentinel and USCHUNT across different UPC patterns. Asterisks (*) denote statistically significant differences based on McNemar's test.

UPC Pattern	**#Active UPCs**	**Accuracy (Active) %**	
		UPC Sentinel	**USCHUNT**
Transparent+ERC-1967	235	232 (98.72%)	**235 (100.0%)**
Unstructured	69	59 (85.51%)	**68 (98.55%)***
Non-pattern	54	**46 (85.19%)***	30 (55.56%)
ERC-1967	49	47 (95.92%)	47 (95.92%)
Inherited	48	**41 (85.42%)**	31 (64.58%)
Inherited+Eternal	33	30 (90.91%)	**33 (100.0%)**
ERC-1822	12	**12 (100.0%)**	9 (75.0%)
Beacon	7	7 (100.0%)	7 (100.0%)
Registry/Beacon	6	**5 (83.33%)**	2 (33.33%)
Registry+Inherited	6	5 (83.33%)	**6 (100.0%)**
Transparent	5	**5 (100.0%)**	4 (80.0%)
Registry	3	**3 (100.0%)**	1 (33.33%)
Transparent+Unstructured	3	3 (100.0%)	3 (100.0%)
Registry+Unstructured	2	0 (0.0%)	**2 (100.0%)**
Diamond	2	2 (100.0%)	2 (100.0%)
Eternal	1	1 (100.0%)	1 (100.0%)
Beacon/Inherited	1	1 (100.0%)	1 (100.0%)
ERC-1822+ERC-1967	1	0 (0.0%)	**1 (100.0%)**
MasterCopy/Singleton	1	0 (0.0%)	0 (0.0%)
Total	**538**	**499 (92.8%)**	**483 (89.8%)**

For the remaining groups, UPC Sentinel and USCHUNT demonstrate comparable accuracy, with no statistically significant differences. In seven categories (ERC-1967, Beacon, Transparent+Unstructured, Diamond, Eternal, Beacon/Inherited, and MasterCopy/Singleton), both systems achieve similar levels of accuracy. UPC

Sentinel performs slightly better than USCHUNT in five categories (Inherited, ERC-1822, Registry/Beacon, Transparent, and Registry), while USCHUNT slightly outperforms UPC Sentinel in another five categories (Transparent+ERC-1967, Inherited+Eternal, Registry+Inherited, Registry+Unstructured, and ERC-1822+ERC-1967). Overall, UPC Sentinel successfully identifies the upgradeability pattern type for 92.8% of UPCs, compared to 89.8% for USCHUNT.

6.8 Discussion

6.8.1 Lessons Learned

In Section 4.2.4.A, we identified 209 contracts in which no `delegatecall` statement could be found at the decompiled bytecode level. A detailed examination of their source code revealed that, in all cases, the proxy functionality was not accessible through the contract's exposed public or external interfaces. This limitation can lead to erroneous detection of proxy contracts when relying solely on source code. Below, we outline the risks associated with source-level proxy detection by describing three patterns of unreachable proxy functionality observed in our analysis of these 209 contracts.

Case #1: Lack of Proxy Exposure. Figure 6.18 presents a simplified[40] UML class diagram of eight contracts listed under the `ProxyDeployer`[41] contract on Etherscan. The `ProxyDeployer` library facilitates the deployment of UPCs by creating

[40] Visit https://etherscan.io/viewsvg?t=1&a=0xa8305b9cfd9590d328e1ac5b13bdd647daec5652 for a detailed class diagram.

[41] https://etherscan.io/address/0xa8305b9cfd9590d328e1ac5b13bdd647daec5652#code

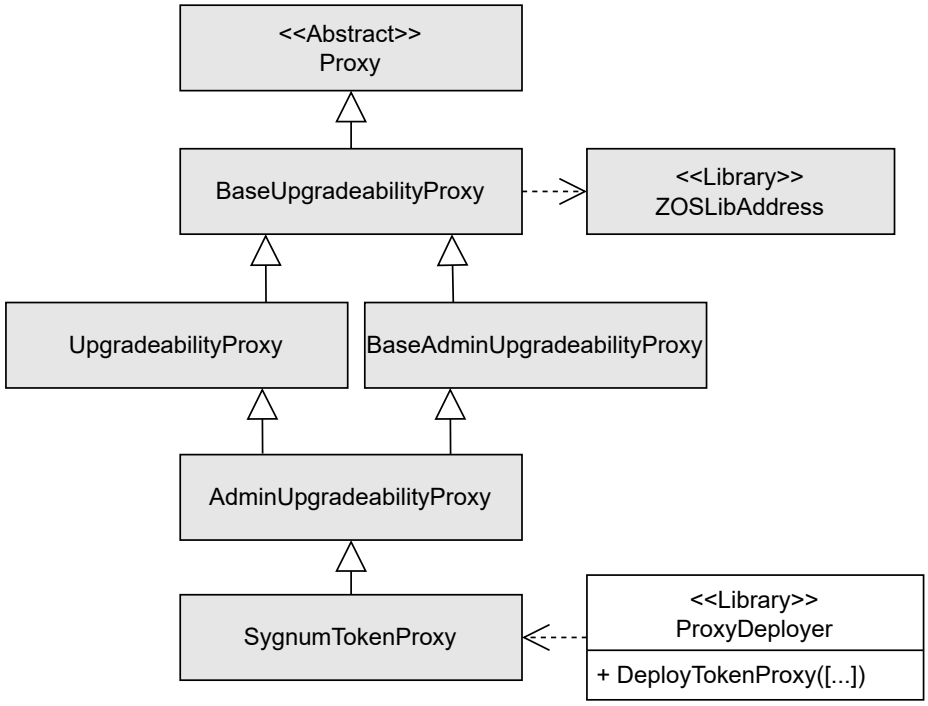


Figure 6.18: Simplified class diagram of the `ProxyDeployer` contract. Contracts with gray backgrounds represent upgradeable proxy contracts.

instances of `SygnumTokenProxy`, a variant of the `AdminUpgradeabilityProxy`, enabling seamless upgrades for Sygnum's tokens. The base contracts, `Proxy` and `BaseUpgradeabilityProxy`, handle call delegation and dynamic implementation updates, while `ZOSLibAddress` ensures the validity of target addresses. `BaseAdmin UpgradeabilityProxy` introduces an admin-only upgrade mechanism, and the `Admin UpgradeabilityProxy` and `UpgradeabilityProxy` contracts establish initialization routines and admin-level control mechanisms.

Although the upgradeability proxy mechanism is indeed present in this example, the `ProxyDeployer` contract itself is not a proxy. The `ProxyDeployer` (i.e., the main

contract[42]) does not expose any public or external function that allows direct access to the proxy functionality (i.e., a violation of Property #1). Instead, `ProxyDeployer` follows the Factory Design Pattern and operates as a deployment factory. Rather than inheriting from the upgradeability infrastructure, it depends on it to instantiate new UPCs on the Ethereum blockchain. Specifically, when the `deployTokenProxy` function of the `ProxyDeployer` is invoked, it deploys a new `SygnumTokenProxy` contract and returns its address.

As a result of this dependency-based architecture, verifying and compiling the `ProxyDeployer` source code requires the full set of dependencies (i.e., all the gray contracts shown in Figure 6.18). This design limits the accessibility of proxy functionality at the source level, which can lead to misclassification if detection relies solely on the presence of `delegatecall` in the top-level contract.

This example illustrates that sequentially scanning (i.e., analyzing one contract at a time) the verified source code of contracts under a given address, without accounting for the semantic relationships between the main contract and its dependencies, can lead to incorrect detection of proxy contracts at the source code level. In contrast, the decompiled bytecode produced by Panoramix does not suffer from this limitation. Specifically, the decompiled bytecode includes all public and external functions exposed by the main contract. For each such function, Panoramix linearizes any cross-contract calls: for example, if a public function `F1` in contract `C1` invokes function `F2` in contract `C2`, Panoramix embeds the logic of `F2` within `F1`

[42] An Ethereum Contract Account (CA) at a specific address can consist of multiple contracts, with one serving as the main contract and others as its dependencies. The main contract integrates and interacts with these supporting contracts, orchestrating the logic, permissions, and data flow. It acts as the front-facing interface of the CA, encapsulating the core functionality and serving as the main point of interaction for users and external entities.

during the decompilation process. Consequently, if `F2` implements proxy logic, its behavior will be reflected within `F1` in the decompiled output.

This transformation enables direct analysis of the decompiled logic of public and external functions to verify the two key properties of a proxy contract (Section 6.2.3). Therefore, analyzing decompiled bytecode offers a more accurate and holistic view of a contract's runtime behavior and interaction patterns than relying solely on the official, verified source code. Out of the 209 instances we examined, 171 exhibited this issue, highlighting that their verified source code was insufficiently reliable for accurate proxy detection.

Takeaway: An Ethereum Contract Account (CA) should only be classified as a proxy if its proxy functionality is exposed through one of the contract's public or external functions. Detecting this at the source code level requires advanced cross-contract analysis to understand the relationships between the main contract and its dependencies. In contrast, Panoramix's decompiled bytecode simplifies these relationships into a linearized format, enabling more reliable proxy detection.

Case #2) Dead proxy code. A dead proxy contract refers to a contract that is neither used by the main contract nor any of its dependencies. Figure 6.19 shows a simplified[43] UML class diagram of the five contracts listed under the `AddressUtils`[44] contract in Etherscan. The `Proxy` contract facilitates delegation to a designated

[43]Visit https://etherscan.io/viewsvg?t=1&a=0xf2262f092063cd19050f7e037cbf6401b58e9035 for a detailed class diagram.

[44]https://etherscan.io/address/0xf2262f092063cd19050f7e037cbf6401b58e9035#code

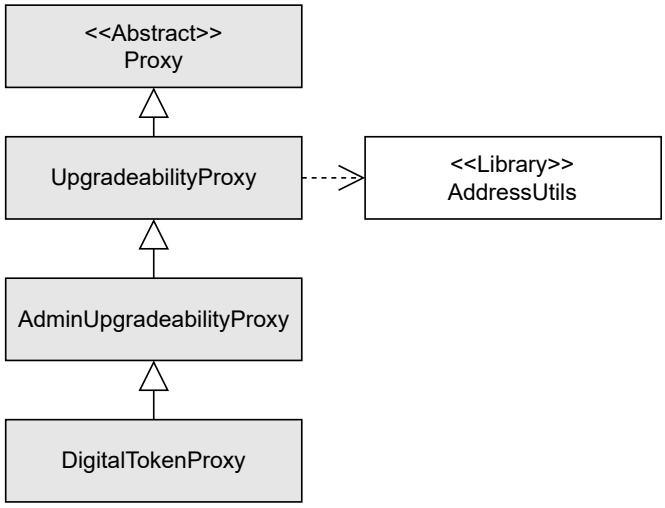


Figure 6.19: Simplified class diagram of the `AddressUtils` contract. Contracts with gray backgrounds represent dead upgradeable proxy contracts.

implementation contract. `UpgradeabilityProxy` extends this functionality by enabling the implementation address to be upgraded. `AdminUpgradeabilityProxy` introduces administrative control, allowing only authorized users to perform upgrades. `DigitalTokenProxy` leverages these mechanisms specifically for digital tokens, using `AdminUpgradeabilityProxy` to support secure and seamless contract upgrades while minimizing disruption for users. Finally, `AddressUtils` is a utility library that provides a function to check whether a given address corresponds to a contract. In this case, although proxy-related contracts exist within the address, they are effectively "dead" since neither the main contract (`AddressUtils`) nor any of its dependencies utilize them in their logic or function calls.

As shown in Figure 6.19, the main contract (i.e., `AddressUtils`) does not depend on any of the other contracts. While it is likely that all five contracts belong to the same software system, one can successfully verify the source code of

`AddressUtils` on Etherscan by submitting only its source code during the verification process. Although the remaining four contracts are not reachable from the main contract, their source codes still appear under the verified source code section of `AddressUtils` on Etherscan. We conjecture that this phenomenon is related to Etherscan's contract verification process, which does not validate whether all submitted contracts are actually referenced or used. In other words, it appears possible to upload dead contracts, i.e., contracts that are not invoked or imported, alongside the relevant ones during the verification process, and Etherscan will display the source code for all of them under the same Contract Account (CA) address once verification is complete.

To validate our hypothesis, we deployed an exact copy of the `AddressUtils`[45] contract to the Sepolia Ethereum testnet, which is supported by Etherscan. We then verified the deployed contract on Etherscan using only the source code for `AddressUtils`, and the verification was successful. Next, we deployed another instance of the `AddressUtils`[46] contract using a similar configuration. This time, however, we uploaded the source code for all five contracts, `Proxy`, `Upgradeability Proxy`, `AdminUpgradeabilityProxy`, `DigitalTokenProxy`, and `AddressUtils`, during the verification process. The verification again succeeded. When we examined the verified source code on Etherscan, we found that all five contracts were listed, even though four of them constitute dead code. This confirms our earlier hypothesis: Etherscan's verification process does not exclude dead contracts from the verified source view if their source files are submitted along with the relevant contract during verification.

[45] https://sepolia.etherscan.io/address/0xe09c93edad08dffc684e8b51da0cd30b9dd50653#code
[46] https://sepolia.etherscan.io/address/0x0f741460ae5f5b8cdbf1ae9181adbf5927fcfc66#code

This example demonstrates that the verified source code under a Contract Account (CA) can contain noise, such as dead contracts, which negatively impacts proxy detection at the source code level. In contrast, contract bytecode is immune to this issue, as the Solidity compiler (`solc`) includes only the main contract and its active dependencies during the compilation process, omitting any unreachable or unused code. Since Panoramix operates on the bytecode, its output remains unaffected by the presence of dead contracts in the verified source. Among the 209 examined instances, 34 were found to contain dead proxy code at the source code level.

Takeaway: Dead contracts included during source code verification, due to Etherscan's lack of validation, can lead to incorrect proxy detection at the source level. This issue is avoided at the bytecode level, as compilers exclude unreachable contracts, ensuring Panoramix receives a clean, noise-free input for reliable decompilation.

Case #3) Overriding proxy functionality. Figure 6.20 shows a simplified[47] UML class diagram of the five contracts listed under the GEOPAY[48] contract on Etherscan. `GEOPAY` is an ERC-20 token contract named "GEO Utility Token". It incorporates safe arithmetic operations via the `SafeMath` contract and implements standard ERC-20 functionalities, such as transferring tokens, approving spenders, and querying balances. The `Ownedby` proxy contract manages ownership while enabling upgradeability through a `fallback` function that delegates calls to the current implementation

[47] Visit https://etherscan.io/viewsvg?t=1&a=0xfa82f0a05b732deaf9ae17a945c65921c28b16dd for a detailed class diagram.

[48] https://etherscan.io/address/0xfa82f0a05b732deaf9ae17a945c65921c28b16dd#code

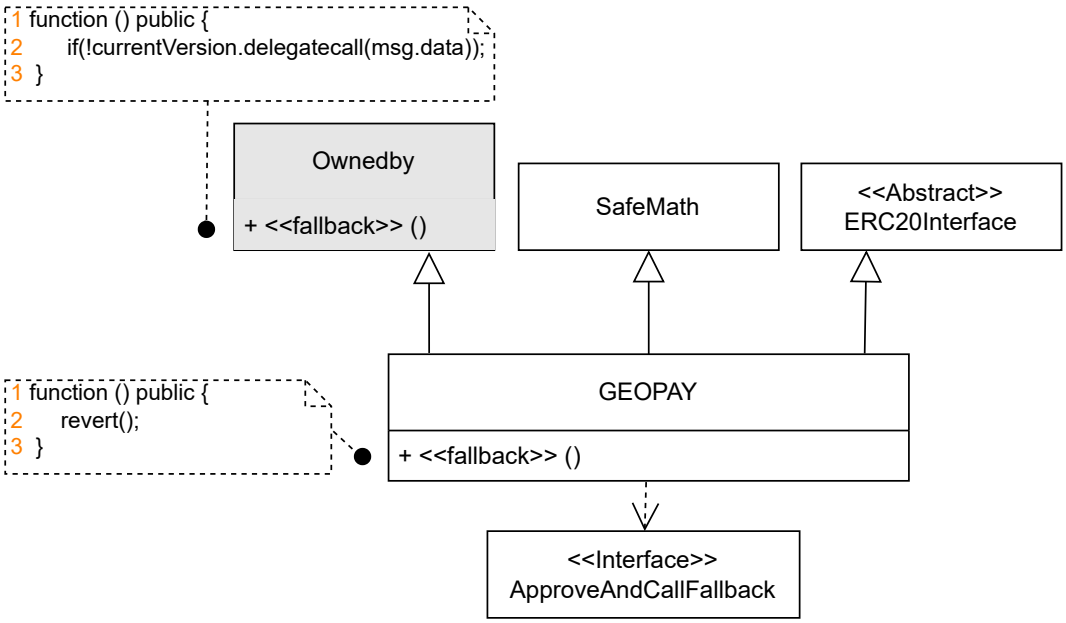


Figure 6.20: Simplified class diagram of the `GEOPAY` contract. Contracts with gray backgrounds represent upgradeable proxy contracts.

contract. Additionally, `GEOPAY` supports an `ApproveAndCall` mechanism for interacting with approved contracts and includes a feature allowing the owner to transfer out accidentally received ERC-20 tokens or ether.

The `GEOPAY` contract inherits from `Ownedby`, whose `fallback` function implements a proxy. However, `GEOPAY` overrides this `fallback`, effectively disabling the proxy functionality. As a result, `GEOPAY` is not a proxy contract. Similar to the previous cases, this example illustrates code that becomes unreachable due to method overriding. This highlights how overlooking object-oriented features, such as method overriding, can lead to incorrect identification of UPCs at the source code

level. In contrast, the Panoramix decompiled bytecode captures such inheritance effects, enabling more accurate detection. Among 209 instances examined, four were affected by this issue.

Takeaway: Method overriding can disable inherited proxy functionality. Researchers should be cautious, as overridden functions may alter or nullify upgradeability behavior in smart contracts.

6.8.2 Applications of UPC Sentinel

Detecting UPCs serves as a gateway to numerous subsequent research avenues. We envision an additional layer on top of our proposed architecture (Section 6.2.2) that leverages insights from the first three layers to support various monitoring tasks for upgradeable proxy contracts. While these applications are not exclusive to our approach, UPC Sentinel enables a more accurate identification of active UPCs, thereby covering a broader subset of all upgradeable proxies.

Prior literature has begun exploring diverse applications related to the evolution, impact, and security of upgradeable proxy contracts. For example, recovering the contract release history addresses the challenge of tracking version changes in smart contracts, given that Etherscan does not natively support links to source repositories or maintain integrated version histories. Qasse et al. (2023) proposed an initial approach to detect new versions through emitted upgrade events, although this method relies on developers consistently implementing and triggering such events. Bodell III et al. (2023) further analyzed upgrade frequency by counting invocations of upgrade functions. Another line of application is evolutionary analysis, which

investigates how smart contracts evolve over time in response to security, efficiency, functionality, and regulatory demands. Qasse et al. (2023) also examined the motivations behind upgrades, such as bug fixes or feature enhancements. Furthermore, researchers have begun evaluating how updates affect user trust, engagement, and asset liquidity. Since blockchain trust depends on perceived immutability and transparent governance, frequent upgrades without sufficient communication can erode user confidence and raise concerns about centralization or hidden risks (Qasse et al. 2023). These studies analyze changes in transaction volume, user retention, user diversity, and asset liquidity associated with the contract (DappRadar 2021; FasterCapital 2024).

A key limitation of the current body of work is its primary reliance on verified (i.e., open-sourced) contracts. This constraint arises because most existing UPC detectors depend on source code availability. However, recent studies show that the majority of smart contracts remain unverified (Xu et al. 2020; Huang et al. 2021; Li et al. 2024a; Chen et al. 2019; Chen et al. 2021c; Chen et al. 2022b), raising concerns that current findings may not generalize across the full spectrum of deployed contracts.

This is where UPC Sentinel offers a significant advantage by operating entirely at the bytecode level. Our empirical evaluation, conducted on two distinct ground truth datasets comprising over 4,000 data points, demonstrates that UPC Sentinel is: (i) effective; (ii) resilient to source-code-level pitfalls (Section 6.8.1); and (iii) capable of accurately detecting active UPCs across a wide variety of known upgradeability proxy patterns, regardless of source code availability. This broader applicability empowers future studies to carry out deeper, more inclusive analyses

that are not constrained by verification status, enabling more comprehensive assessments of upgradeability practices throughout the Ethereum ecosystem. Accordingly, UPC Sentinel could serve as an autonomous monitoring tool or be integrated with platforms like Etherscan to continuously track and report changes in UPC implementation contracts.

6.8.3 Release History Recovery Using UPC Sentinel

Unlike conventional software, where version control systems like Git offer a clear history of versions and updates, Ethereum lacks an integrated mechanism for tracking smart contract versioning. Platforms such as *Etherscan* do not provide an explicit release history, despite this information being indirectly available on-chain. Moreover, Etherscan does not link deployed contracts to their corresponding GitHub repositories, making it difficult to trace a contract's evolution. Although Qasse et al. (2023) propose detecting upgrades based on emitted events, this approach relies on developers consistently implementing and triggering such events, which cannot be guaranteed.

To address this gap, we envision extending UPC Sentinel to recover the release history of smart contracts. For example, Figure 6.21 illustrates our proposed process for reconstructing the release timeline of an upgradeable proxy contract C.[49]

First, UPC Sentinel's proxy detection module identifies C as a proxy contract (Lines 23–30) and detects three distinct implementation contracts (Imp#1, Imp#2, Imp#3) that C has ever delegated calls to, sorted by deployment time.

[49] https://etherscan.io/address/0x2f3f840d17eb61020680c1f4b00510c3caa7df63#code

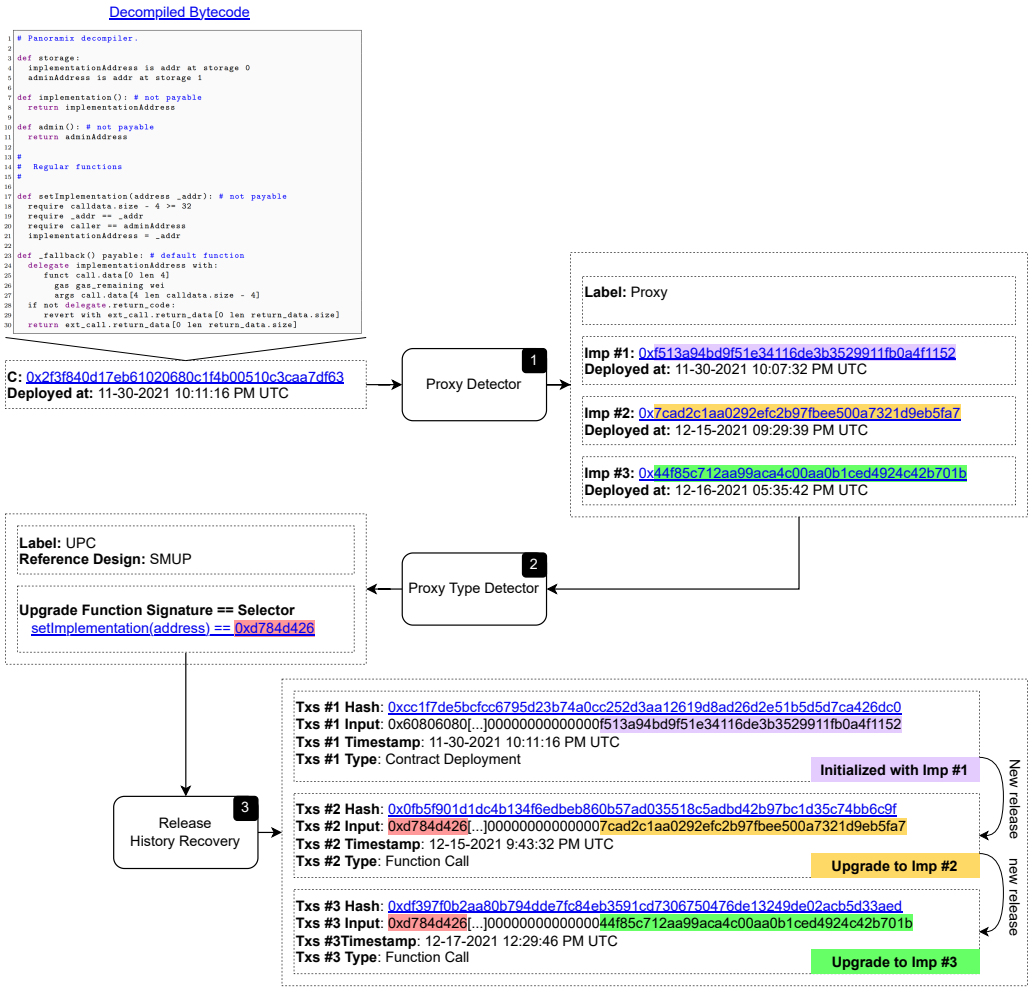


Figure 6.21: Recovering a contract's release history using UPC Sentinel. Highlighted colors indicate the locations where the upgrade function selector and distinct implementation contract addresses are identified within the retrieved transactions.

Second, the proxy type detection layer analyzes the decompiled bytecode of C to determine whether it is an upgradeable proxy contract (UPC) or Forwarder

Proxy Contract (FPC). It identifies C as an SMUP-based UPC and extracts the upgrade function signatures and selectors responsible for assigning new implementation contract address. In this case, the function `_setImplementation(address)` with selector `0xd784d426` is identified as the upgrade function (Lines 17–21).

Third, using the outputs from the previous steps, we recover the contract's release history by collecting: (i) the deployment transaction of C, and (ii) all C's inbound transactions whose `calldata` begins with `0xd784d426`, indicating a call to the upgrade function.

By sorting these transactions chronologically, we can infer the release sequence. For instance, Imp#1 is initially assigned through the constructor (Tx#1). Later, Tx#2 updates the implementation to Imp#2 via the upgrade function, followed by Tx#3, which replaces it with Imp#3, yielding a release history: Imp#1 $\rightarrow$ Imp#2 $\rightarrow$ Imp#3.

Finally, we acknowledge that the proposed method is limited to recovering the release history of *active*, upgradeable contracts that follow recognized UPC patterns (Section 2.15.6). If a contract employs multiple upgrade strategies, such as initially using the `selfdestruct` method and later migrating to a proxy-based mechanism, our method cannot reconstruct the complete release history. Furthermore, in cases involving the Diamond Upgradeability Proxy, where a proxy delegates calls to multiple facet contracts, a more sophisticated approach would be required to accurately associate upgrades with the corresponding facets. We leave this challenge as an open direction for future work.

6.8.4 Time-Cost of UPC Sentinel

Our experiments were conducted on a high-performance computing system equipped with 60 CPU cores, 500 GB of RAM, and a network bandwidth of 90 Mbps, ensuring efficient handling of large-scale data processing tasks and fast data transfer. For data retrieval, we utilized the free tier of Google BigQuery to query the `crypto_ethereum` dataset. In the following sections, we discuss the time-cost of running UPC Sentinel and the optimizations implemented to improve its time efficiency.

Time-cost of initializing the bytecode decompiler. UPC Sentinel's upgradeability detector layer operates on the decompiled bytecodes generated by the Panoramix decompiler for input proxy contracts, their implementation contracts, and their dependency contracts. Given the high prevalence of type-1 clones (i.e., contracts with identical bytecodes) in the Ethereum ecosystem, this component incorporates a memory mechanism to avoid redundant decompilation of contracts sharing identical bytecodes. Additionally, it supports multi-core processing (set to 60 cores in this study) and enforces a decompilation timeout parameter (set to 60 minutes per input contract), allowing multiple contracts to be decompiled in parallel. We leveraged this multi-core capability across all relevant UPC Sentinel components to accelerate processing. For instance, if a proxy contract has several implementation contracts, all of them are decompiled concurrently rather than sequentially. To further accelerate and reduce the cost of on-the-fly bytecode retrieval, this component precomputes a bytecode hash for all Ethereum contracts using a BigQuery SQL query. The resulting table is automatically downloaded locally and converted into a

mapping of `<contract address: string, bytecode hash: string>`. Additionally, another mapping is created between *distinct* bytecodes and their corresponding hashes (`<bytecode hash: string, bytecode: string>`). Once initialized, these two mappings allow any contract's bytecode to be retrieved in $\mathcal{O}(1)$ time. The entire initialization process, including data retrieval, download, and loading, took 13.6 minutes.

Time Cost of Initializing the Bytecode Decompiler. We conducted a detailed time-cost analysis of running UPC Sentinel on the GE (Section 6.4) and GB (Section 6.6) ground truth datasets, which contain 3,177 and 920 contracts, respectively. Specifically, we broke down the total execution time for each dataset into four key components: (i) proxy detector initialization, (ii) bytecode decompiler initialization, (iii) total decompilation time, and (iv) the time required to execute UPC Sentinel's detectors. These metrics are summarized in Table 6.7, providing an in-depth understanding of how each component contributes to the overall runtime. Lastly, we report the average processing time per contract. Since initialization occurs only once, we also report the average per-contract time excluding the initialization times for both the proxy detector and the bytecode decompiler.

Time-cost of running UPC Sentinel on the GE & GB datasets. We conducted a time-cost analysis of running UPC Sentinel on the GE (Section 6.4) and GB (Section 6.6) ground truth datasets, which contain 3,177 and 920 contracts, respectively. Specifically, we break down UPC Sentinel's total execution time for each dataset into four components: proxy detector initialization time, bytecode decompiler initialization time, total decompilation time, and the time taken to execute UPC Sentinel's detectors. These results are presented in Table 6.7, allowing us to

assess the contribution of each component to the overall runtime. Finally, we report the average computation time per contract. Since initialization is performed only once, we also provide the average computation time per contract excluding the initialization times for the proxy detector and bytecode decompiler.

Table 6.7 presents the time cost of running UPC Sentinel across the two datasets. For the GE ground truth dataset, which includes 3,177 contracts, the entire process took 4.29 hours to complete, resulting in an average processing time of 4.9 seconds per contract. Similarly, for the GB ground truth dataset containing 920 contracts, the process took 2.06 hours, with an average processing time of 8 seconds per contract. When excluding the initialization time for the proxy detector and bytecode decompiler, the adjusted average processing time was reduced to 4.2 seconds for the GE dataset and 5.9 seconds for the GB dataset.

Compared to USCHUNT, which has an average processing time of 0.76 seconds per contract, UPC Sentinel's processing time is significantly higher, with an average time cost that is 7.8 times greater $\left(\frac{5.9}{0.76}\right)$. We identified decompilation as the most time-consuming component across both datasets. To optimize this, we plan to pre-decompile all Ethereum contracts as a future enhancement. This approach is feasible because, despite the deployment of over 50 million contracts on Ethereum as of September 2022, only 1.33% have *distinct* bytecodes. By pre-decompiling these *distinct* bytecodes, we can eliminate the need for on-the-fly decompilation and retrieve the decompiled version of any contract in $\mathcal{O}(1)$ time. Under this scenario, the total processing time for the GB dataset would be reduced to 13.78 minutes, averaging 0.89 seconds per contract, closely aligning with USCHUNT's performance.

Table 6.7: Breakdown of UPC Sentinel's time cost across GE and GB datasets.

Dataset	#Contracts	Proxy Detector Init Time (min)	Decompiler Init Time (min)	Decompilation Time (hr)	Detectors Time (min)	Total Execution Time (hr)	Avg Time per Contract (sec)	Avg Time per Contract Excluding Init. (sec)
GE	3,177	20.2	13.6	3.51	12.85	4.29	4.9	4.2
GB	920	20.2	13.6	1.27	13.78	2.06	8.0	5.9

Alternatively, employing a faster decompiler is another viable strategy to reduce the overall time cost.

6.9 Threats to Validity

In this section, we discuss potential threats to the validity of our findings, covering construct, internal, and external validity dimensions.

6.9.1 Construct Validity

In terms of our limitations, UPC Sentinel currently cannot detect inactive (upgradeability) proxy contracts. Ebrahimi et al. (2023) analyzed a statistically representative random sample of 385 contracts drawn from the entire set of Ethereum contracts, including both open-source and closed-source ones. Their study found that 90 (23.4%) of these contracts were proxy contracts, of which 62 (16.1%) were active and 28 (7.27%) were inactive. However, since inactive proxies are not necessarily obsolete (e.g., they may be activated in the future or upgraded in response to evolving needs), UPC Sentinel can be periodically applied to monitor inactive cases over time. Hence, such inactive proxies can be identified immediately after they are used for the first time. Additionally, we propose a future improvement plan to extend UPC Sentinel for more proactive detection of inactive (upgradeability) proxy contracts.

Proxy detection layer: Our method in Section 6.2.3 currently uses the two conceptual properties of a proxy contract and assesses contracts' transactions against them to detect *active* proxy contracts. However, due to the lack of transactional activity for inactive proxy contracts, this method is ineffective for the latter type. To support inactive proxy contracts, future studies can leverage insights we obtained during our study regarding the potential detection of inactive proxies based solely on decompiled bytecode.

In particular, we found information equivalent to the two conceptual properties of proxy contracts (Section 6.2.3) at the decompiled bytecode level. Regarding Property #1, one can reliably determine whether the decompiled bytecode of a given contract contains a function with a `delegatecall` statement by searching for

the “delegate” keyword. For Property #2, one can assess the decompiled bytecode of the function where the “delegate” keyword appears using our three identified lexical patterns (Figure 6.5), to verify whether the function maintains an interface similar to that of its implementation contract. In Section 6.2.4, these lexical patterns were present in 99.8% of a representative sample of active proxy contracts. As a result, transaction activity may no longer be important for detecting proxy contracts, given these two equivalent properties in the decompiled bytecode. However, given this study’s focus on active proxy contracts, we leave the investigation of inactive proxy contracts as a promising avenue for future work.

Proxy type classification layer: For each of our three upgradeability reference designs, we explain how one could adapt the detection process to support inactive proxies. For SMUP-based UPCs, once an inactive proxy is identified by the improved proxy detection layer discussed above, no further changes are needed to the A: SMUP Detector module (Figure 6.4), as there are no design differences between inactive and active UPCs of this type.

For DUP-based UPCs, where the upgradeability logic is implemented inside the proxy’s implementation contracts, our algorithm (Figure 6.7) requires the list of the proxy’s implementation contract addresses. This list is currently obtained by analyzing the proxy’s historical transactions (i.e., dynamic analysis) for active proxy contracts (Section 6.2.3). However, this approach proves ineffective for inactive proxy contracts due to the lack of transactional data. To address this, one can obtain the latest implementation contract address of an inactive proxy contract by reading the proxy’s storage. Specifically, one could run a static analyzer to extract the storage slot of the implementation variable from the decompiled bytecode of

an inactive proxy contract, then use a blockchain node service provider like Infura to read the value of this slot (i.e., the implementation contract address) directly from the proxy's storage. This approach could be effective, as the implementation contract address is typically initialized by developers using the proxy's constructor or via an initializer function after deployment.

Regarding ESUP-based UPCs, our method (Figure 6.6) relies on analyzing historical transactions to obtain the target contract address (i.e., the external contract that the proxy queries to fetch the current implementation contract address right before delegating calls). To handle this for inactive cases, one can locate the storage slot where the external contract address is stored in the proxy's storage and use an Infura node to read its value. The rest of our algorithms for the DUP and ESUP detectors would remain unchanged for inactive cases.

To ensure that our evaluations are not influenced by the presence of duplicates, we identified and removed repeated instances of the same contract address in the ground truth dataset (Section 6.3.2.3). Assuming two contracts C1 and C2 in the ground truth, there are two interpretations of duplicates: (1) C1 and C2 have identical bytecodes but are deployed at different Ethereum contract addresses; (2) C1 and C2 have identical Ethereum contract addresses.

We removed only duplicates under interpretation (2), as under interpretation (1), C1 and C2 still have distinct addresses (i.e., different entities) and their own unique runtime metadata and state (e.g., bytecodes, transactions, implementation contracts, storage values). For instance, they may differ in their implementation contracts or dependencies, which makes them distinct input contracts. However, with interpretation (2), both contracts, addresses, bytecodes, runtime transactions,

are completely identical, effectively representing the same deployed contract. In the ground truth dataset provided by Bodell III et al. (2023), we identified 40 instances where the same contract address appeared multiple times (i.e., interpretation (2)). By retaining only one unique entry per contract address, we ensured that each contract is represented only once, thereby preventing any skewing of results due to duplicate data points.

In our study, we opted not to use Cohen's Kappa to measure inter-rater agreement due to concerns about its reliability with imbalanced datasets. Research by Deloitte (2023) and Li et al. (2023a) highlights Kappa's sensitivity to category prevalence, which often leads to unstable or misleading results when one category is either very rare or very common, even when there is high observed agreement between raters.

This issue arose in RQ1, where the first and third authors (i.e., the raters) independently verified the upgradeability status of a sample of 94 contracts. This sample was drawn from a dataset of 3,177 contracts flagged as UPC by Etherscan, where the positive category (UPC class) was expected to be highly prevalent. As a result, Kappa behaved counterintuitively, yielding a disproportionately low value despite substantial agreement between raters (in 92 (97.87%) out of 94 cases). A similar scenario occurred in RQ3, where we compared Bodell III et al. (2023)'s ground truth upgradeability labels (i.e., first rater) with our assigned labels (i.e., second rater) across four groups (i.e., FN, TN, TP, FP). Since one category (UPC or Non-UPC) was significantly overrepresented in each group, the category imbalance rendered Kappa unreliable. To address these limitations, we instead reported

the observed proportion of agreement (McHugh 2012), which directly reflects rater consistency without being skewed by category prevalence.

Furthermore, the Kappa statistic is effective when the positive classification rate is balanced, i.e., raters are equally likely to classify subjects as positive or negative (approximately 50%). However, when one rater significantly favors one category, the Kappa measure becomes unreliable (Gwet 2002; Byrt et al. 1993). This issue arose in RQ2, where we compared Bodell III et al. (2023)'s ground truth proxy labels (i.e., first rater) with our verified proxy labels (i.e., second rater). In this case, all 920 analyzed contracts in the ground truth were labeled as proxy, and our analysis determined 702 contracts to be proxy. This extreme imbalance rendered the Kappa statistic unreliable, resulting in a zero value despite the relatively high agreement between our labels and the ground truth. To address this, we reported the observed proportion of agreement (McHugh 2012), which directly reflects the consensus between our verified labels and the ground truth.

6.9.2 Internal Validity

While evaluating decompilers is outside the scope of our current research, we acknowledge that Panoramix may not always complete the decompilation process or may produce inaccurate decompiled code, as evidenced by our experiments and corroborated by a recent study (Liu et al. 2023a). However, evidence suggests that the impact of such cases on our results is not significant.

Specifically, Liu et al. (2023a) conducted an empirical study on five state-of-the-art smart contract decompilers, including Erays (Zhou et al. 2018), Vandal (Brent

et al. 2018), Panoramix (Palkeo 2019), EthervmDec[50], and Gigahorse (Grech et al. 2019). They analyzed decompilation success rates for these tools on a representative random sample of 16,000 smart contracts. Their results show that, except for Erays, the remaining four decompilers achieved high success rates (all above 98%), with Panoramix reaching 98.2%. Furthermore, Panoramix generated contract sources with the highest similarity scores to original sources, crucial for detecting upgradeability proxy contracts accurately at the decompiled bytecode level. By contrast, Erays and Gigahorse generated code with near-zero similarity scores, complicating the UPC detection process. These findings indicate that Panoramix is superior in terms of code similarity, which is a key reason for selecting it in this study.

In our experiments (specifically RQ1), we were unable to determine the type of 22 (0.7%) out of 3,177 UPCs due to decompilation failures. Similarly, in RQ3, one (0.1%) out of 920 contracts featured an inconsistent storage structure due to Panoramix failing to recognize the storage pattern. Despite this, the impact of decompilation issues was minimal, with less than 1% of UPC instances not detected across two different analyses. Overall, we believe our methodology and upgradeability reference designs (Section 6.2.4) offer a comprehensive roadmap for detecting UPCs. We encourage researchers to replicate our study using other decompilers to further validate our findings and enhance the robustness of this research area.

Finally, we conducted a thorough assessment of both ground truth datasets (GE and GB), including systematic sanity checks to validate their quality. Following established data quality guidelines (Bhatia et al. 2024b), we defined clear, well-justified criteria for dataset modifications. Each adjustment was documented in

[50] https://ethervm.io/decompile

detail, including rationale and supporting evidence, adhering to empirical software engineering principles to promote transparency (Ralph et al. 2021). Furthermore, all results and findings have been made publicly accessible through our replication package to support peer review, reproducibility, and future research.

6.9.3 External Validity

Our collected ground truth dataset (GE) comprises active UPCs deployed from January 2018 through September 2022, as identified by Etherscan. This dataset is particularly valuable as it reflects verified UPCs from nearly every month within this timeframe, as shown in Figure 6.12. The dataset's consistency and broad temporal coverage suggest it potentially captures overall trends in UPC usage during this period. The steady rise in UPCs, especially from 2020 onward, also aligns with the increasing ratio of EOAs deploying proxy contracts (Ebrahimi et al. 2023). That said, we acknowledge that this dataset includes only UPCs for which at least two versions are reported by Etherscan. Such UPCs may follow specific storage standards, making them easier for Etherscan to identify and track. While this may introduce bias, since not all UPCs follow these standards, we believe the insights derived from this dataset are still valuable for evaluating UPC Sentinel.

To further mitigate bias, we also used another ground truth dataset from a prior study (Bodell III et al. 2023) and evaluated UPC Sentinel on it. This dataset includes contracts initially identified as UPCs by Slither and refined by Bodell III et al. (2023). One potential limitation is that all contracts in this ground truth are verified (i.e., their source code is available on Etherscan), whereas most Ethereum contracts are not verified. Hence, this ground truth may not allow complete recall

measurement for UPC Sentinel or USCHUNT. Nonetheless, preparing a balanced dataset is highly resource-intensive, and among prior studies, only Bodell III et al. (2023) evaluated their method using a ground truth. Thus, by using two different ground truths, one of which we compiled, we ensure that our evaluation meets or exceeds the rigor of existing studies in the field.

Finally, although our dataset may not be fully representative of true negative cases, we believe UPC Sentinel handles true negatives (i.e., contracts that are not upgradeability proxy contracts) effectively for two reasons. First, our proxy detection method achieves 100% precision, thereby excluding numerous true negatives with perfect accuracy. Second, our upgradeability detection mechanism is based on the fundamental features of the most common proxy patterns (i.e., our upgradeability reference designs). This systematic design ensures determinism and reliability, reducing the likelihood of false negatives, which is confirmed by our perfect precision in Section 6.6. Nevertheless, our findings may not generalize to other programmable blockchain platforms.

6.10 Conclusion

The growing complexity and widespread adoption of decentralized applications have underscored the critical need to reconcile blockchain immutability with the practical necessity for updates. Proxy contracts, particularly upgradeability proxy contracts (UPCs), have emerged as a leading solution to this challenge, offering a balance between flexibility and efficiency. Despite their increasing prevalence and importance, existing methods for detecting UPCs face several limitations, particularly in handling closed-source contracts, covering a wide range of patterns, and

efficiently combining static and dynamic analysis techniques. These gaps highlight the pressing need for more effective tools to detect and classify UPCs.

This chapter was driven by the hypothesis that, despite challenges such as closed-source code, diverse UPC variants, and complex cross-contract interactions, it is feasible to accurately distinguish Upgradeable Proxy Contracts (UPCs) from Forwarder Proxy Contracts (FPCs) and their variants directly at the bytecode level. In response, we introduced UPC Sentinel, a three-tier algorithm that systematically orchestrates both static and dynamic analysis techniques on decompiled smart contract bytecode to identify UPCs. Each layer of UPC Sentinel is designed with clearly defined techniques to fulfill its objective with high precision. Given a smart contract address: (i) the proxy detector layer accurately determines whether the contract is a proxy by analyzing its transactional behavior against the two inherent properties of proxy contracts; (ii) the upgradeability detector layer combines static and dynamic analyses to assess whether the proxy supports upgradeability, based on compliance with three abstract upgradeability reference designs; and (iii) the pattern classifier layer provides fine-grained classification by identifying the specific upgradeability pattern the detected UPC implements.

Our comprehensive empirical evaluation, conducted on two distinct ground truth datasets, demonstrates that UPC Sentinel is both effective and reliable within the defined scope of this study, namely, active proxy contracts. On our newly collected GE dataset containing 3,177 UPCs, UPC Sentinel achieved a near-perfect accuracy of 99%. Similarly, on the GB dataset, it reached a perfect precision of 100% and a near-perfect recall of 99.3%, outperforming state-of-the-art methods (Table 6.5). These results validate our initial hypothesis, proving that high-accuracy

classification is achievable at the bytecode level. Importantly, UPC Sentinel does not rely on source code availability, a limitation of many prior approaches, enabling it to detect active UPCs at the bytecode level across a wide variety of upgradeability proxy patterns and their variants. This capability supports a more comprehensive analysis of upgradeability practices in the Ethereum ecosystem, including closed-source contracts.

While our study focused on active proxy contracts, UPC Sentinel may require future adaptations to effectively handle inactive proxy contracts. We have outlined concrete plans to extend UPC Sentinel to support the detection of such cases. Ultimately, we hope our methodology and replication package, including the publicly available source code of UPC Sentinel, ground truth datasets, and detailed results, will serve as a valuable resource for the research community, fostering further progress in this domain and enabling others to enhance, extend, and build upon our work.

CHAPTER 7

Conclusion

THIS chapter concludes the thesis by synthesizing the research conducted, summarizing the key findings, and discussing their broader implications. It revisits the initial objectives and research questions, outlines the principal contributions to knowledge, acknowledges the limitations of the study, and proposes avenues for future research.

7.1 Scope

The central aim of this thesis was to investigate the tension between two competing forces in the Ethereum ecosystem: the immutability of smart contracts, which guarantees trust, transparency, and resistance to unauthorized modification, and

the practical need for smart contract evolution. This investigation was guided by the following overarching thesis hypothesis:

H1. Technical debt is significantly common in Ethereum smart contracts.

H2. Proxy contracts constitute a significant proportion of deployed Ethereum smart contracts.

H3. Despite challenges such as closed-source code, diverse UPC variants, and complex cross-contract interactions, it is feasible to accurately distinguish Upgradeable Proxy Contracts (UPCs) from Forwarder Proxy Contracts (FPCs) and their variants directly at the bytecode level.

The first part of the thesis, presented in Chapter 4, is dedicated to validating hypothesis H1. It moves beyond anecdotal evidence to conduct a rigorous, large-scale empirical study on technical debt (TD) in Ethereum smart contracts. This chapter establishes TD as a critical, yet understudied, counterpart to the extensive research on unintentional security vulnerabilities. It investigates how this deliberate debt accumulates and manifests in Solidity code, analyzing its types, severity, and evolution over time to provide a data-driven assessment of its significance and commonality in the ecosystem.

The second part of the thesis, presented in Chapter 5, is dedicated to validating hypothesis H2. Given Ethereum's immutability, addressing quality issues like TD after deployment is inherently challenging, necessitating built-in mechanisms for evolution. This chapter focuses on the proxy pattern as the primary enabler of this evolution. It characterizes the proxy landscape, providing a large-scale analysis

to quantify the adoption of proxy-based patterns and confirm their significant proportion within the deployed contract population. This establishes the foundational context for the deeper investigation into their complexity in the subsequent chapter.

Building on the findings of H2, the final part of the thesis, presented in Chapter 6, addresses the core technical challenge posed by hypothesis H3. It confronts the practical problems of reduced transparency and complex interactions inherent to proxy systems. The chapter details the development and validation of a novel toolchain designed to accurately distinguish between Upgradeable Proxy Contracts (UPCs) and Forwarder Proxy Contracts (FPCs) directly from their bytecode. By successfully overcoming the challenges of closed-source code, diverse implementation variants, and intricate call patterns, this work demonstrates the feasibility of precise, bytecode-level proxy classification, a crucial capability for auditors and analysts.

In summary, this thesis scope progresses from establishing the prevalence of quality issues (H1), to quantifying the architectural solution designed to address them (H2), and finally to solving a key technical problem in analyzing that solution (H3), thereby providing a comprehensive examination of software engineering challenges in the immutable Ethereum environment.

7.2 Key Findings

7.2.1 Chapter 4

Chapter 4 investigates the nature, prevalence, and ecosystem-level implications of Self-Admitted Technical Debt (SATD) in Ethereum smart contracts. SATD was

identified through developer comments in Solidity source code, capturing instances where developers explicitly acknowledge technical shortcomings, deferred tasks, or limitations. A large-scale exploratory analysis was conducted across thousands of real-world contracts to examine the prevalence of SATD and its correlation with contract usage and value inflow. The study also examined the role of code cloning in propagating technical debt and included a qualitative analysis that led to a novel SATD taxonomy tailored to smart contract development.

Our findings indicates that while SATD was found in only 1.5% of contracts, it was highly concentrated in critical DApps, namely, contracts processing millions of transactions and significant financial inflows. Remarkably, 93.5% of SATD-bearing contracts were identified as clones, with over 84% clustered in just ten clone groups, this indicates that technical debt is not randomly distributed, but rather propagates structurally through code reuse and automated deployment pipelines. These findings are significant because they reveal that SATD in Ethereum is less about individual developer negligence and more about systemic factors and conscious trade-offs in an immutable, decentralized environment. By empirically answering key research questions, this chapter demonstrates that SATD is both technically embedded and socially reinforced. These insights lay the foundation for future SATD-aware development tools, policy changes in contract reuse, and smarter strategies for managing debt in software systems that cannot be retroactively patched. The chapter concludes by reflecting on the practical implications of SATD in immutable and decentralized environments. It offers insights for researchers interested in smart

contract maintainability, as well as tool builders aiming to assist developers in identifying, managing, or repaying SATD in a domain where upgrades are costly and challenging.

7.2.2 Chapter 5

Chapter 5 presents the first large-scale exploratory study of proxy contracts in Ethereum. Motivated by the critical need for a scalable and accurate method to identify proxies, without relying on heuristics, source code, or transaction replaying, which are significant limitations of prior work, we introduced a high-precision method to detect active proxy contracts. We then applied our method to a comprehensive dataset of over 50 million contracts and 1.6 billion transactions to investigate the practical prevalence of proxy contracts in Ethereum from multiple angles. Next, we analyzed the architectural decisions behind proxy deployment, as these creational patterns, though largely unstudied, have profound implications for system scalability, cost-efficiency, and auditability. Furthermore, we examined the primary engineering motivations for adopting proxies by distinguishing their fundamental purpose: whether they are used to enable contract evolution (i.e., Upgradeable Proxy Contracts) or to achieve other objectives such as cost-efficient cloning and modularity (i.e., Forwarder Proxy Contracts). Finally, to assess the maturity and risk profile of the ecosystem, we evaluated the extent to which developers adhere to established reference implementations, thereby shedding light on the adoption of best practices versus the continued use of potentially riskier custom designs.

Our findings underscore the central role of the proxy pattern in Ethereum's contract architecture, revealing that over 14% of all deployed contracts are proxies. We identified 12 distinct creational patterns for deploying proxies, which we grouped into on-chain and off-chain styles. On-chain deployments, typically using factory contracts, dominate the ecosystem, accounting for 99.3% of all proxies and 60.9% of all use cases, and are especially suited for DApps that require multi-instance proxy deployment at scale. In contrast, off-chain styles are simpler and more common in singleton use cases but offer limited auditability and automation. We found that proxies are more frequently employed for other purposes (67.8%, classified as Forwarder Proxy Contracts) than for enabling contract evolution (32.2%, classified as Upgradeable Proxy Contracts). This reflects Ethereum's economic constraints and the widespread adoption of gas-optimized deployment strategies, particularly minimal proxies like ERC-1167, which alone account for 29.4% of all use cases. Overall, nearly 80% of detected use cases conform to established reference designs, indicating potential adherence to community standards. However, the remaining 21% employ custom implementations which highlights further investigation. This chapter concludes by offering actionable design guidelines and trade-off analyses for developers, along with practical implications and open questions for researchers. Importantly, our qualitative study revealed that nearly one-third of proxies are designed to support contract evolution, setting the stage for the next chapter.

7.2.3 Chapter 6

Chapter 6 builds directly on the findings of Chapter 5, shifting focus from general proxy detection to the more nuanced problem of identifying and classifying

Upgradeable Proxy Contracts (UPCs), the dominant mechanism for enabling smart contract evolution on Ethereum. While the previous chapter introduced a scalable technique for detecting active proxy contracts and explored their deployment architectures and engineering motivations, this chapter addresses a critical follow-up question: Which proxies are actually designed to support contract evolution? To answer this, we introduce UPC Sentinel, a novel three-layer algorithm that operates directly on smart contract bytecode. UPC Sentinel overcomes key limitations of prior work, including dependency on the availability of source code, limited coverage of known UPC patterns, and the inability to analyze multi-contract (transitive) dependencies. UPC Sentinel employs a hybrid static-dynamic analysis framework to (i) detect proxy contracts, (ii) classify them as Upgradeable Proxy Contracts or Forwarder Proxy Contracts, and (iii) further classify detected UPCs into eight known UPC patterns (Section 2.15.6).

We conduct an extensive empirical evaluation of UPC Sentinel using two curated ground truth datasets (i.e., GE and GB) with over 4000 labeled data points. We benchmark our method against USCHUNT, a state-of-the-art source-level detector. On the GE dataset of 3,177 known UPCs, UPC Sentinel achieved 99% recall, correctly identifying 3,147 contracts, including complex cases involving deep transitive dependencies. On the GB dataset, UPC Sentinel consistently outperformed USCHUNT, a state-of-the-art source-level detector, across both mixed and active-only settings. In the mixed setting, UPC Sentinel achieved F1-scores of 90.5% (proxy) and 78.1% (non-proxy), compared to USCHUNT's 86.5% and 0.0%. In the active-only setting, UPC Sentinel reached perfect F1-scores of 100.0% for both classes, while USCHUNT scored 84.1% (proxy) and 0.0% (non-proxy). For proxy

type classification, UPC Sentinel again outperformed USCHUNT. In the mixed setting, it achieved F1-scores of 90.0% (UPC) and 81.6% (non-UPC), compared to USCHUNT's 86.1% and 51.2%. In the active-only setting, UPC Sentinel achieved near-perfect scores of 99.6% (UPC) and 99.2% (non-UPC), while USCHUNT lagged behind with 83.8% and 50.7%, respectively, demonstrating UPC Sentinel's superior accuracy and robustness across detection and classification tasks. In UPC pattern classification, both methods achieved high accuracy overall (92.8% for UPC Sentinel vs. 89.8% for USCHUNT), with complementary strengths. USCHUNT performed better on Unstructured patterns, while UPC Sentinel excelled in identifying UPCs that did not conform to known reference patterns, highlighting its generalizability to real-world scenarios. These differences were statistically significant, validating the superior coverage and precision of UPC Sentinel in bytecode-level analysis. The chapter concludes with key insights into why source-level detection can fail, due to existence of unreachable dead proxy functionality, and emphasizes the reliability and generalizability of bytecode-based analysis. UPC Sentinel's design enables accurate, comprehensive, and source-independent classification of evolving smart contracts and offers a practical method for ecosystem monitoring, audit support, and longitudinal studies of contract evolution and upgradeable patterns on Ethereum.

7.3 Thesis Contributions

This thesis makes three overarching contributions to the study of Ethereum smart contracts and their evolution:

1. Self-Admitted Technical Debt (SATD) in Smart Contracts: We conduct the first large-scale empirical study on technical debt in Ethereum, showing that despite immutability, TD is actively incurred and even propagated through code cloning and tooling. We develop a smart contract-specific TD taxonomy, reveal practical and ecosystem-level implications, and provide critical insights and data to guide future tooling and developer practices.

2. Understanding and Characterizing Proxy Contracts: Through the first comprehensive analysis of proxy contracts at scale, we reveal their growing prevalence and evolution across 50 million contracts and 1.6 billion transactions. We identify 12 proxy deployment patterns and provide empirical insights into their trade-offs, risks, and benefits, especially regarding transparency, modularity, governance, and contract evolution. We also release four ground truth datasets to support reproducible proxy contract research.

3. UPC Sentinel: A Robust Upgradeability Proxy Contract Detection Algorithm: We propose UPC Sentinel, a three-layer bytecode-level detection algorithm that distinguishes proxy contracts, classifies their types (e.g., UPC vs. FPC), and identifies specific UPC patterns. Backed by rigorous design, curated datasets, and extensive evaluation, UPC Sentinel significantly outperforms source-level approaches, especially in closed-source scenarios, and enables deeper analysis of smart contract evolution. All artifacts have been publicly released to support transparency and future benchmarking.

Collectively, these contributions provide an integrated empirical and technical foundation for understanding, detecting, and reasoning about maintainability, modularity, and upgradeability in Ethereum smart contracts. We provide a detailed list of our contribution in Chapter 1.

7.4 Practical Implications

7.4.1 For Developers

For developers, this thesis underscores the importance of proactively designing smart contracts with evolution in mind. Given Ethereum's immutability and the presence of SATD, particularly in critical, high-traffic DApps, developers should integrate evolution mechanisms, such as proxy patterns, early in the architecture. Doing so enables safer and more manageable evolution without disrupting users or requiring complex migration procedures. The study also highlights recurring SATD anti-patterns, including hard-coded values and unbounded operations, which can be avoided through better design choices and static analysis. Developers are encouraged to adopt existing solutions and lean on tooling support, such as Hardhat or Foundry plugins, that flag problematic code constructs during development.

Moreover, developers often reuse third-party contracts or deploy contracts cloned from existing frameworks or DApps, many of which contain SATD. As shown in our findings, a large portion of SATD-bearing contracts are systematically propagated through automated deployment pipelines or reused libraries. To avoid incurring the same technical debt, developers should inspect reusable contracts for SATD before integration, for instance, using our four SATD indicator patterns (`TODO`,

`WORKAROUND`, `FIXME`, and `HACK`) and avoid blindly cloning contract templates without understanding debt implications. If reuse is necessary, developers should either isolate the SATD-bearing component behind an upgradeable proxy contract or adapt the reused contract to mitigate or remove known issues. Documentation should clearly note inherited debt limitations so that future maintainers and front-end developers are aware of the inherited debt. Additionally, once a SATD item has been addressed or refactored, developers should remove the corresponding SATD comment to prevent confusion or misinterpretation.

The decision to adopt the proxy pattern should be the result of a careful trade-off analysis. While proxy contracts offer clear benefits, such as enabling seamless contract evolution, promoting modularity, and reducing deployment costs through logic reuse, they also introduce substantial architectural and security complexities. As discussed in Section 5.8.1, developers must weigh these advantages against known drawbacks, including increased runtime gas overhead, heightened risk of misconfiguration (e.g., storage collisions, admin key compromise), and reduced transparency due to indirection between the proxy and its implementation. These trade-offs are especially critical in high-value or long-lived DApps, where auditability, maintainability, and user trust are paramount.

Beyond the decision to use proxies, developers must also evaluate how proxies are deployed, off-chain or on-chain. Each style carries distinct implications for scalability, governance, and system transparency. Off-chain deployments, while simple and suitable for small-scale or singleton contracts, suffer from limitations in auditability, reuse, and automation. On-chain deployments, by contrast, provide scalable and collaborative deployment workflows with improved traceability, batch

deployment support, and deterministic address generation (via `CREATE2`), albeit at the cost of added design complexity and initial gas expense. Section 5.8.2 provides a detailed comparison, and to support informed decision-making, this thesis offers a structured decision flowchart (Figure 5.17) that guides developers through deployment choices based on project size, need for predictability, multi-user access, and governance constraints.

7.4.2 For Ecosystem Platforms

Tool builders are well-positioned to improve smart contract transparency and maintainability by automating the detection and annotation of both technical debt and proxy patterns. At present, the discovery of self-admitted technical debt (SATD) is limited to manual inspection of Solidity comments, despite our findings that over 10,000 real-world contracts, including high-value and high-traffic deployments, contain such debt. Tools integrated into blockchain explorers like Etherscan could greatly enhance transparency by flagging contracts with common SATD indicators (e.g., `TODO`, `FIXME`), displaying context-aware warnings, and annotating risk levels. Our proposed SATD taxonomy provides a structured and domain-specific framework to categorize and interpret these instances, enabling tools to not only detect but also contextualize the impact of different types of SATD, such as gas-inefficiency, non-future-proof logic, or Solidity workarounds.

In parallel, current capabilities for proxy detection in Etherscan are limited and largely heuristic-driven. By contrast, UPC Sentinel leverages bytecode-level analysis to accurately detect and classify proxy contracts, including Upgradeable Proxy Contracts (UPCs) and their specific pattern variants, even in the absence of verified

source code. This bytecode-centric approach allows for more robust, scalable, and generalizable detection across the Ethereum ecosystem. Tool builders can further extend such detection systems to support downstream tasks, including grouping proxies into use cases, mapping proxy contracts to all their logic contracts, and tracking upgrade events over time.

Importantly, the lack of transparency around proxy–logic links, as revealed in our analysis (Observation 2), where nearly all DApps with proxies failed to publish associated logic contract addresses, presents a significant ecosystem-wide gap. DApp marketplaces such as DappRadar, StateOfTheDApps, or Dapp.com can adapt our detection methodology for integration into their platforms to automatically identify whether a DApp utilizes (upgradeable) proxy contracts, and if so, to link those proxies to their corresponding logic contracts. This integration would not only improve transparency and auditability for end users but also enrich the metadata and completeness of marketplace listings. Ultimately, UPC Sentinel provides a foundation upon which developers, auditors, and platform maintainers can build more trustworthy, observable, and verifiable smart contract ecosystems. In this way, tool builders act as a critical bridge, operationalizing empirical research findings into actionable ecosystem-wide infrastructure for transparency, safety, and long-term maintainability.

7.4.3 For Auditors

An auditor is a security professional or specialized firm that performs a comprehensive, independent review of a smart contract's code and architecture. For these stakeholders, this thesis highlights several critical challenges and opportunities in

maintaining trust and accountability in Ethereum smart contract systems. First auditors must consider the presence and implications of self-admitted technical debt (SATD). SATD can reflect deferred bug fixes, unoptimized code paths, or temporary workarounds that, if left unaddressed, may introduce performance issues, security risks, or user-facing failures. Since SATD is typically buried in source comments and often overlooked in traditional audits, its systematic assessment is an emerging responsibility. Our SATD taxonomy provides auditors with a structured lens for evaluating the severity, type, and potential impact of technical debt, particularly in critical or reusable contracts. By flagging SATD alongside functional and security issues, auditors can offer more holistic assessments of contract quality and maintainability.

In addition to technical debt risks, proxy-based architectures require auditors to move beyond single-contract analysis and instead examine entire contract systems, including proxies, logic contracts, and upgrade interfaces, as tightly coupled components. The risks associated with such designs, including storage collisions, admin key misuse, and function selector clashes, demand rigorous inspection protocols and a solid understanding of bytecode-level behavior. Because source-level analysis may include dead unreachable proxy contract code snippets that does not reflect runtime behavior, auditors should prioritize methods that operate at the bytecode level to avoid false positives and ensure complete coverage. UPC Sentinel, for example, offers a source-independent and more reliable foundation for detecting and classifying upgradeable proxies across real-world deployment scenarios.

Finally, this thesis underscores the importance of transparency. In our analysis (Observation 2), nearly all DApps that used proxies failed to disclose their associated logic contracts, undermining auditability and increasing systemic opacity. Auditors are uniquely positioned to advocate for better practices, including full publication of proxy–logic pairs, the adoption of established upgradeability standards, and the use of observable governance mechanisms (e.g., multisig). Through these efforts, auditors play a central role in upholding trust, detecting hidden liabilities, and safeguarding users within Ethereum ecosystems.

7.4.4 For Smart Contract Users

Our findings have several implications for end users, who often lack the technical knowledge to assess smart contract risks. One concern is that proxy contracts, especially those with undisclosed or centralized upgrade paths, can undermine trust in DApps. Users interacting with such systems should be aware that they may not be engaging with a stable, immutable contract, and that functionality or fund behavior may change after an upgrade. The lack of visibility into associated logic contracts can obscure potential vulnerabilities, backdoors, or governance issues. Similarly, SATD buried in contract comments may reflect temporary workarounds, gas inefficient functionality, or unfinished features, all of which could impact user experience. When SATD is not documented or disclosed in front-end code, users may encounter unexpected failures, higher transaction fees, or degraded performance.

To mitigate these risks, users are encouraged to rely on platforms that flag contract debt and contract evolution, and to favor DApps that adopt robust governance

models, such as multi-signature schemes or time-locked upgrades, over those governed by a single privileged entity. However, this places a burden on the ecosystem to develop more accessible and user-centric tooling. Building on top of UPC Sentinel, future tools can present users with intelligible summaries of a contract's upgradeability status, the history of logic changes, and the transparency of its governance process. Such tools could integrate into blockchain explorers or DApp marketplaces, enabling users to make more informed decisions about which contracts they choose to trust with their assets or interactions. Ultimately, user awareness, supported by ecosystem tooling, is essential to maintaining confidence, usability, and safety in a decentralized environment.

7.4.5 For Researchers

For researchers, this thesis opens multiple avenues for further inquiry and methodological advancement. First, the shift from monolithic to modular smart contract architectures demands a transition toward system-level analysis (Observation 1). Researchers should develop new frameworks to identify, map, and analyze interdependent contracts into systems (e.g., use cases) rather than focusing solely on standalone code. Second, the findings on SATD and proxy patterns provide a rich empirical foundation for further exploring trends in technical debt propagation, contract evolution, and upgrade governance practices. UPC Sentinel, by operating entirely on bytecode, enables researchers to conduct broader, source-independent

studies, extending the reach of prior work constrained to verified contracts. This capability is particularly critical given that most contracts on Ethereum remain closed-source, making source-level analysis insufficient for comprehensive ecosystem-wide studies.

Furthermore, researchers have an essential role in systematically analyzing the risks associated with proxy smart contracts at the bytecode level. This includes uncovering patterns of misuse (e.g., uninitialized contracts, access control issues, storage layout collisions, function selector clashes, unsafe `selfdestruct` usage, incorrect `delegatecall` context, reentrancy during upgrades, upgrade lock-in, stealth upgrades, poor version tracking, initialization ordering problems, fake proxies), measuring the real-world prevalence of unsafe upgrade practices, and evaluating the transparency of deployed DApps (Feist et al. 2019; Salehi et al. 2022; Bodell III et al. 2023; Ruaro et al. 2024; Chen et al. 2024; Zhang et al. 2025a; Huang et al. 2024). UPC Sentinel provides the foundational infrastructure for such future investigations by accurately identifying (upgradeable) proxy contracts and classifying them into known pattern variants, even in the absence of source code.

A particularly important research direction involves the analysis of governance mechanisms for contract evolution. In this context, governance refers to the process through which smart contract changes are authorized, whether centrally by a single party or collectively through stakeholder participation (Palladino 2020; Salehi et al. 2022). Even the most technically robust upgrade mechanism is rendered unsafe without proper governance; a single misused admin key or opaque upgrade path can undermine the very trustlessness smart contracts aim to provide. Researchers should study the adoption and effectiveness of various governance techniques (e.g.,

Externally Owned Accounts (EOAs), multi-signature schemes, timelocks, pausability, escape hatches, commit-reveal upgrade flows, and on-chain voting mechanisms) to assess their alignment with the scale, value, and complexity of DApps (Palladino 2020). UPC Sentinel makes this analysis feasible by reliably identifying upgradeable proxy contract and enabling future studies to further assess the underlying governance mechanisms of such contracts.

With this capability, researchers can define actionable risk metrics, construct release engineering taxonomies, and characterize governance models across diverse contract ecosystems. These insights can be operationalized by tool builders and auditors to support real-time monitoring, automated compliance checks, and public accountability. As such, UPC Sentinel outputs can seed the development of accessible, open-source tools and datasets, such as visual dashboards, upgradeability maps, and contract evolution timelines, that empower users and enrich DApp marketplaces and block explorers. Ultimately, UPC Sentinel enables future researchers to perform more comprehensive and accurate analyses of proxy-based smart contracts.

7.5 Limitations

While each core chapter of this thesis (Chapters 4,5, and 6) includes a detailed discussion of chapter-specific threats to validity, here we reflect on broader limitations that apply across the thesis as a whole.

First, this thesis focuses exclusively on the Ethereum blockchain and leverages datasets that are time-bounded, up to March 2019 for SATD and September 2022

for proxies. While Ethereum remains the dominant platform for smart contract development, its design and tooling may not reflect those of other platforms such as BNB Smart Chain, Polygon, or Avalanche. Similarly, given the fast pace of ecosystem evolution, newer development practices, proxy standards, and technical debt strategies may not be captured in our analyses.

Second, the detection methods used in Chapters 4 and 5 rely on heuristic-based and behavior-based techniques grounded in prior work. In Chapter 4, SATD detection was conducted using Jitterbug, a pattern-matching tool that identifies debt-related comments containing explicit keywords such as `TODO`, `FIXME`, `HACK`, or `WORKAROUND`. While effective for capturing clearly marked instances, this approach may miss hard-to-find SATDs, comments that imply technical debt but do not use these standard indicators, such as vague limitations or undocumented design trade-offs. In Chapter 5, proxy detection is based on analyzing transaction history to identify runtime delegation behavior. This approach ensures high precision and avoids false positives; however, it cannot detect inactive proxies whose delegation logic has not yet been triggered. Based on our findings, such inactive proxies represent approximately 7.3% of contracts in a representative sample, suggesting that the actual prevalence of proxy usage may be slightly underestimated.

Third, the UPC Sentinel tool developed in Chapter 6 is optimized for high-precision detection of active upgradeable proxy contracts at the bytecode level. However, its current implementation does not fully support inactive proxies or less common upgradeability patterns not represented in our evaluation datasets. Although we outline potential extensions, such as leveraging decompiled bytecode and storage inspection, these enhancements remain avenues for future work.

Finally, although this thesis contributes new datasets and tooling, we have not yet conducted real-world validation through user studies, developer interviews, or IDE integration. As a result, the practical utility and adoption of our techniques remain untested in live development environments.

7.6 Future Work

While the discussion sections of Chapters 4,5, and 6 each identify chapter-specific limitations and suggest some targeted future research directions, this section highlights several broad opportunities for future work that emerge from the thesis as a whole.

7.6.1 SATD

Our detection approach employs Jitterbug's easy-to-find component, a pattern recognition method designed to identify explicit instances of SATD based on well-established textual patterns. This method is efficient, scalable, and performs well in detecting clearly marked SATD, as demonstrated by its low false-negative rate (0.5%) and manageable false-positive rate (13.9%). 17 out of the total 40 false posties stemmed from outdated comments, cases where the debt had already been addressed, highlighting a broader challenge in distinguishing active from resolved SATD. While effective for detecting easy-to-find SATD, our employed approach is limited in its ability to uncover hard-to-find cases (0.5%) such as those expressed through vague, unconventional, or implicit language. Future work could explore the application of machine learning or large language models to address this gap. Such models could

learn from semantic and contextual cues in developer comments and code structure, enabling the detection of subtle or non-standard SATD instances and further improving the comprehensiveness and reliability of SATD analysis in smart contracts.

For instance, Listing 7.1 presents a case of hard-to-find technical debt in the `TetherToken` contract, a cryptocurrency designed to maintain a 1:1 peg with the U.S. dollar, which has a market value of $157.53B and a transaction count exceeding 247 million. Specifically, line 17 shows this debt. The presence of the `onlyPayloadSize` modifier in this contract exemplifies a subtle form of technical debt. Originally introduced as a workaround for the ERC-20 short address attack, this modifier checks the length of `msg.data` to prevent malformed inputs. However, the underlying vulnerability has since been addressed in Solidity version 0.5.0 and later, which enforce stricter ABI validation and type checking. Consequently, the modifier has become obsolete, it introduces unnecessary complexity, reduces code readability, and may mislead developers into believing it still provides meaningful protection. This outdated, technology-induced workaround reflects deeper issues such as legacy code, obsolete safeguards, and increased maintenance overhead.

That said, a promising direction for future work is to systematically study this subset of SATD, *technology-induced debt*, to understand its root causes, persistence, and impact on long-term maintainability. In particular, the Solidity ecosystem presents a compelling target due to its rapid evolution. Future research could build automated classifiers to detect such instances across contracts and link them to issue trackers, release notes, or documentation of language and framework changes.

This could inform both tool developers, by surfacing critical pain points, and practitioners, by identifying legacy code segments that may no longer be necessary as the ecosystem matures.

```
pragma solidity ^0.4.17;
[...]
/**
 * @title Basic token
 * @dev Basic version of StandardToken, with no allowances.
 */
contract BasicToken is Ownable, ERC20Basic {
   using SafeMath for uint;

   mapping(address => uint) public balances;

   // additional variables for use if transaction fees ever became necessary
   uint public basisPointsRate = 0;
   uint public maximumFee = 0;

   /**
   * @dev Fix for the ERC20 short address attack.
   */
   modifier onlyPayloadSize(uint size) {
      require(!(msg.data.length < size + 4));
      _;
   }
   [...]
}
[...]
```

Listing 7.1: TetherToken Contract

Moreover, while our findings establish a strong link between SATD and code cloning, further investigation is beneficial to distinguish between intentional propagation and accidental replication of technical debt. This distinction is important: intentional reuse of SATD-laden code may reflect conscious trade-offs due to time

constraints or limited alternatives, whereas accidental replication points to hidden debt propagation risks that may silently accumulate across contracts. Understanding this nuance could inform more intelligent tooling capable of flagging cloned debt instances based on context, and could guide developers in prioritizing remediation efforts.

In addition, the SATD taxonomy developed in this study, though grounded in empirical observations, should be validated and refined through user studies with developers, auditors, and tooling experts to ensure practical relevance and completeness. Practitioner feedback can help uncover overlooked or emerging categories of debt and improve the taxonomy's alignment with real-world needs. Such validation would also enhance the adoption of SATD-aware analysis tools by ensuring that their outputs match the mental models and expectations of those maintaining and auditing smart contracts.

7.6.2 Proxy Detection

While our detection approach achieves perfect precision and recall for active proxy contracts, it does not capture *inactive proxies*, which account for approximately 7.27% of cases in our ground truth dataset. These are contracts that implement the proxy pattern but have not yet triggered any transactional behavior involving `delegatecall`. To address this limitation, future work could explore hybrid methods that integrate static bytecode analysis with runtime behavioral patterns. In particular, the two inherent properties of proxy contracts can be systematically detected via decompiled bytecode (e.g., using Panoramix), as demonstrated in our evaluation (Section 6.2.4.2.1). Further research could advance this line of inquiry

by leveraging symbolic execution and analyzing raw bytecode to identify potential proxy structures independent of execution traces and decompilation overhead. As a complementary solution, periodically re-running our proxy detection method on new transactional data can help surface proxy contracts that were previously inactive but have since become active. This incremental analysis approach would allow researchers and auditors to maintain an up-to-date view of the proxy contract landscape and track activation trends over time (Section 6.9.

In traditional software systems, release engineering practices rely heavily on accessible version control systems, changelogs, and source code diffs to understand evolution across versions. However, in the Ethereum ecosystem, such artifacts are often absent or unverifiable, particularly for unverified contracts. As a result, studying contract evolution requires a paradigm shift: from source-level analysis to direct bytecode-level understanding. Our work contributes to this shift by introducing UPC Sentinel, which accurately identifies proxy-based evolution mechanisms even in the absence of source code. Also, our proposed release recovery method (Section 6.8.3) enables the reconstruction of historical upgrade events by tracking changes to the logic contract over time, thus providing a foundation for studying smart contract evolution directly at the bytecode level. Building on this, bytecode-level approaches such as STAN (Li et al. 2020) and SmartBT (Xiang et al. 2025), which generates natural language descriptions of bytecode instructions, offer a bridge toward understanding contract evolution when source code is missing. Additionally, methods like SmartBugBERT (Bu et al. 2025) and COBRA (Li et al. 2024b) demonstrate that bytecode-based vulnerability detection is viable and effective. Such techniques could be extended to scan multiple contract versions

to identify vulnerabilities introduced or mitigated through evolution. Overall, future tools that combine bytecode-based version tracking, vulnerability analysis, and natural language summarization could greatly improve our ability to audit and understand smart contract release practices, particularly in settings where source-level studies (Bodell III et al. 2023; Qasse et al. 2024) often fall short.

Another direction for future work is to study how contract evolution impacts reputation, user engagement, and asset liquidity in Ethereum (DappRadar 2021; FasterCapital 2024). While the proxy pattern enables seamless evolution, its effect on user trust and behavioral dynamics remains unclear. Building on our release recovery approach (Figure 6.21), future studies could analyze transactional patterns across versions, e.g., inbound volume, retention, success rate, and user diversity, as well as asset-level signals like trading volume and price volatility. However, such analyses must carefully control for confounding factors, including market conditions, and external events, to isolate the impact of evolution itself.

Another important avenue for future work is to systematically characterize the contract evolution practices. While this study focuses on the accurate detection and classification of UPCs, broader insights could be gained by analyzing release practices in terms of their frequency, duration between versions, and historical trends in UPC adoption since Ethereum's inception. This includes examining which UPC patterns (Section 2.15.6) were prevalent at different points in time and identifying which designs are currently most popular for implementing upgradeable systems. *UPC Sentinel* provides the essential infrastructure for such investigations by enabling precise identification of UPCs and eight known UPC pattern types, even in the absence of verified source code. Building on this, future work could aim to

map detected UPCs to their corresponding decentralized applications (DApps), thus establishing a foundation for ecosystem-wide transparency.

Furthermore, as the proxy pattern continues to evolve, there is a growing need to maintain up-to-date ground truth datasets for validating detection methods. Most existing studies tend to overlook **Property #2** of the proxy pattern, *interface preservation* between the proxy and its logic contract, which is essential for distinguishing true proxy contracts from superficially similar contract structures (Section 2.15.4). Moreover, source-based ground truths often suffer from critical limitations, such as the presence of dead or unreachable proxy-related code that are not part of the runtime operational bytecode (Section 6.8.1). These issues can lead to inflated false positives and reduce the reliability of source code-level detection methods. To address these gaps, future efforts should focus on constructing community-shared benchmark datasets, enriched with labeled examples of confirmed proxy contracts, their upgrade trajectories, and interface metadata. Such datasets would enable standardized, accurate evaluation and spur further innovation in proxy detection techniques.

Finally, we envision adapting UPC Sentinel for integration into blockchain explorers (e.g., Etherscan), auditing frameworks, and decentralized governance tools. This would empower users and auditors with bytecode-level insights into upgradeability risks, even in the absence of verified source code. Realizing this vision will require collaboration with tool builders (e.g., DApp marketplaces, Etherscan) and attention to usability, scalability, and real-time analytics.

7.7 Closing Remarks

The research presented in this thesis investigates a fundamental and enduring dilemma in Ethereum smart contract development: the conflict between immutability, a pillar of decentralized trust, and the necessity for evolution, adaptation, and maintainability in long-lived software systems. Through empirical analyses, systematic classification, and tool development, this thesis sheds light on how technical debt accumulates and propagates within Ethereum, how developers pragmatically circumvent immutability through proxy-based architectures, and how such patterns can be rigorously detected and analyzed at scale.

By introducing novel methods such as UPC Sentinel and conducting two large-scale studies across tens of millions of contracts, this thesis moves beyond theoretical discussion and into actionable insight. It provides developers, auditors, tool builders, and researchers with new perspectives and concrete artifacts to navigate the complexities of evolving smart contracts in an immutable environment. The findings challenge the conventional view that immutability implies stasis, enforcing instead that smart contracts can, and do, evolve, albeit through intricate and risk-prone mechanisms.

Despite the limitations and rapidly evolving nature of the ecosystem, this thesis establishes a solid empirical and technical foundation for future inquiry and tooling. It invites the community to rethink contract evolution not as an exception to immutability, but as a carefully engineered capability that demands transparency, governance, and architectural discipline. Ultimately, this thesis contributes to a

broader understanding of how to engineer, maintain, and evolve trustworthy decentralized applications in practice, not merely as static artifacts of code, but as living systems embedded within a dynamic socio-technical landscape.

Bibliography

0xAA, vectorized, and 0age (Sept. 2023). *ERC-7511: Minimal Proxy Contract with PUSH0*. Accessed: 03-March-2025. URL: https://eips.ethereum.org/EIPS/eip-7511.

Abdi, Siamak, Giuseppe Di Fatta, Atta Badii, and Giancarlo Fortino (2025). *Blockchain Epidemic Consensus for Large-Scale Networks*. arXiv: 2508.02552 [cs.DC]. URL: https://arxiv.org/abs/2508.02552.

Agarwal, Aman (Apr. 2023). *Proxy contract*. Accessed: 03-March-2025. URL: https://medium.com/geekculture/proxy-contract-7f05ff50a2c7.

AL Amri, Shaima, Leonardo Aniello, and Vladimiro Sassone (Mar. 2023). "A Review of Upgradeable Smart Contract Patterns based on OpenZeppelin Technique". In: *The Journal of The British Blockchain Association*. DOI: 10.31585/jbba-6-1-(3)2023.

Alves, Nicolli S.R., Thiago S. Mendes, Manoel G. de Mendonça, Rodrigo O. Spínola, Forrest Shull, and Carolyn Seaman (2016). “Identification and management of technical debt: A systematic mapping study”. In: *Information and Software Technology* 70, pp. 100–121. DOI: https://doi.org/10.1016/j.infsof.2015.10.008.

Alves, Nicolli S.R., Leilane F. Ribeiro, Vivyane Caires, Thiago S. Mendes, and Rodrigo O. Spínola (2014). “Towards an Ontology of Terms on Technical Debt”. In: *2014 Sixth International Workshop on Managing Technical Debt*, pp. 1–7. DOI: 10.1109/MTD.2014.9.

Antonino, Pedro, Juliandson Ferreira, Augusto Sampaio, and A. W. Roscoe (2022). “Specification is Law: Safe Creation and Upgrade of Ethereum Smart Contracts”. In: *Software Engineering and Formal Methods*. Ed. by Bernd-Holger Schlingloff and Ming Chai. Cham: Springer International Publishing, pp. 227–243. URL: https://arxiv.org/abs/2205.07529.

Aragon (2023). *Aragonos Intro · aragon developer portal*. Accessed: 03-March-2025. URL: https://hack.aragon.org/docs/aragonos-intro.html.

Arcelli Fontana, Francesca, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino (June 2016). “Comparing and experimenting machine learning techniques for code smell detection”. In: *Empirical Softw. Engg.* 21.3, pp. 1143–1191. DOI: 10.1007/s10664-015-9378-4.

Atarpara, JT Riley, Thomas, xiaobaiskill, and Vectorized (Aug. 2024). *ERC-7760: Minimal Upgradeable Proxies*. Accessed: 03-March-2025. URL: https://eips.ethereum.org/EIPS/eip-7760.

Audius Blog (2022). *Audius Governance Takeover Post-Mortem*. Accessed: 03-March-2025. URL: https://blog.audius.co/article/audius-governance-takeover-post-mortem-7-23-22.

Babushkin, Konstantin (Mar. 2024). *Challenges and Pitfalls for Beginner Front-End Blockchain Developers*. Accessed: 03-March-2025. URL: https://maddevs.io/writeups/challenges-for-beginner-front-end-blockchain-developers.

Bavota, Gabriele and Barbara Russo (2016). "A large-scale empirical study on self-admitted technical debt". In: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR '16. Austin, Texas: Association for Computing Machinery, pp. 315–326. DOI: 10.1145/2901739.2901742. URL: https://doi.org/10.1145/2901739.2901742.

Becze, Martin and Hudson Jameso (Oct. 2015). *EIP-1: EIP Purpose and Guidelines*. Accessed: 03-March-2025. URL: https://eips.ethereum.org/EIPS/eip-1.

Behnke, Rob (2021). *he PAID Network Hack*. Accessed: 03-March-2025. URL: https://www.halborn.com/blog/post/explained-the-paid-network-hack-march-2021.

Bez, Mirko, Giacomo Fornari, and Tullio Vardanega (2019). "The scalability challenge of ethereum: An initial quantitative analysis". In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pp. 167–176. DOI: 10.1109/SOSE.2019.00031.

Bhatia, Aaditya, Foutse Khomh, Bram Adams, and Ahmed E Hassan (2024a). *An Empirical Study of Self-Admitted Technical Debt in Machine Learning Software*. arXiv: 2311.12019 [cs.SE]. URL: https://arxiv.org/abs/2311.12019.

Bhatia, Aaditya, Dayi Lin, Gopi Krishnan Rajbahadur, Bram Adams, and Ahmed E. Hassan (2024b). *Data Quality Antipatterns for Software Analytics*. arXiv: 2408.12560 [cs.SE]. URL: https://arxiv.org/abs/2408.12560.

Bloom, Gedare, Bassma Alsulami, Ebelechukwu Nwafor, and Ivan Cibrario Bertolotti (2018). "Design patterns for the industrial Internet of Things". In: *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*, pp. 1–10. DOI: 10.1109/WFCS.2018.8402353.

Bodell III, William E, Sajad Meisami, and Yue Duan (Aug. 2023). "Proxy Hunting: Understanding and Characterizing Proxy-based Upgradeable Smart Contracts in Blockchains". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, pp. 1829–1846. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/bodell.

Bosu, Amiangshu, Anindya Iqbal, Rifat Shahriyar, and Partha Chakraborty (2018). "Understanding the motivations, challenges and needs of Blockchain software developers: a survey". In: *Empirical Software Engineering* 24, pp. 2636–2673. URL: https://api.semanticscholar.org/CorpusID:53278520.

Brent, Lexi, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz (2018). *Vandal: A Scalable Security Analysis Framework for Smart Contracts*. arXiv: 1809.03981 [cs.PL]. URL: https://arxiv.org/abs/1809.03981.

Bu, Jiuyang, Wenkai Li, Zongwei Li, Zeng Zhang, and Xiaoqi Li (2025). *SmartBugBert: BERT-Enhanced Vulnerability Detection for Smart Contract Bytecode*. arXiv: 2504.05002 [cs.CR]. URL: https://arxiv.org/abs/2504.05002.

Byrt, Ted, Janet Bishop, and John B. Carlin (1993). “Bias, prevalence and kappa”. In: *Journal of Clinical Epidemiology* 46.5, pp. 423–429. DOI: https://doi.org/10.1016/0895-4356(93)90018-V. URL: https://www.sciencedirect.com/science/article/pii/089543569390018V.

Center Chris Brummer, Agnes N. Williams Professor of Law at Georgetown University Law (June 2022). “Disclosure, Dapps and DeFi”. In: *Stanford Journal of Blockchain Law & Policy*. URL: https://stanford-jblp.pubpub.org/pub/disclosure-dapps-defi.

Ceriani, Lidia and Paolo Verme (Sept. 2012). “The origins of the Gini index: extracts from Variabilità e Mutabilità (1912) by Corrado Gini”. In: *The Journal of Economic Inequality* 10.3, pp. 421–443. DOI: 10.1007/s10888-011-9188-x.

Chainalysis Team (2022). *Lessons from the Wormhole Exploit*. Accessed: 03-March-2025. URL: https://www.chainalysis.com/blog/wormhole-hack-february-2022/.

Chakraborty, Partha, Rifat Shahriyar, Anindya Iqbal, and Amiangshu Bosu (2018). “Understanding the software development practices of blockchain projects: a survey”. In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM ’18. Oulu, Finland: Association for Computing Machinery. DOI: 10.1145/3239235.3240298. URL: https://doi.org/10.1145/3239235.3240298.

Chen, Cheng-Kang, Wen-Yi Chu, Muoi Tran, Laurent Vanbever, and Hsu-Chun Hsiao (2024). *Proxion: Uncovering Hidden Proxy Smart Contracts for Finding Collision Vulnerabilities in Ethereum*. arXiv: 2409.13563 [cs.CR]. URL: https://arxiv.org/abs/2409.13563.

Chen, Jiachi, Xin Xia, David Lo, and John Grundy (Dec. 2021a). “Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum”. In: *ACM Trans. Softw. Eng. Methodol.* 31.2. DOI: 10.1145/3488245.

Chen, Jiachi, Xin Xia, David Lo, and John Grundy (Dec. 2021b). “Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum”. In: *ACM Trans. Softw. Eng. Methodol.* 31.2. DOI: 10.1145/3488245.

Chen, Jiachi, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen (July 2022a). “DefectChecker: Automated Smart Contract Defect Detection by Analyzing EVM Bytecode”. In: *IEEE Transactions on Software Engineering* 48.7, pp. 2189–2207. DOI: 10.1109/tse.2021.3054928. URL: http://dx.doi.org/10.1109/TSE.2021.3054928.

Chen, Jiachi, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen (July 2022b). “DefectChecker: Automated Smart Contract Defect Detection by Analyzing EVM Bytecode”. In: *IEEE Transactions on Software Engineering* 48.7, pp. 2189–2207. DOI: 10.1109/tse.2021.3054928. URL: http://dx.doi.org/10.1109/TSE.2021.3054928.

Chen, Jiachi, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen (2022c). “Defining Smart Contract Defects on Ethereum”. In: *IEEE Transactions on Software Engineering* 48.1, pp. 327–345. DOI: 10.1109/TSE.2020.2989002.

Chen, Jiachi, Xin Xia, David Lo, John Grundy, and Xiaohu Yang (2020). *Maintaining Smart Contracts on Ethereum: Issues, Techniques, and Future Challenges*. arXiv: 2007.00286 [cs.SE].

Chen, Jiachi, Xin Xia, David Lo, John Grundy, and Xiaohu Yang (Nov. 2021c). "Maintenance-related concerns for post-deployed Ethereum smart contract development: issues, techniques, and future challenges". In: *Empirical Softw. Engg.* 26.6. DOI: 10.1007/s10664-021-10018-0. URL: https://doi.org/10.1007/s10664-021-10018-0.

Chen, Qiuyuan, Chunyang Chen, Safwat Hassan, Zhengchang Xing, Xin Xia, and Ahmed E. Hassan (Apr. 2021d). "How Should I Improve the UI of My App? A Study of User Reviews of Popular Apps in the Google Play". In: *ACM Trans. Softw. Eng. Methodol.* 30.3. DOI: 10.1145/3447808. URL: https://doi.org/10.1145/3447808.

Chen, Qiuyuan, Chunyang Chen, Safwat Hassan, Zhengchang Xing, Xin Xia, and Ahmed E. Hassan (May 2021e). "How should I improve the UI of My App?: a study of user reviews of popular apps in the Google Play". English. In: *ACM Transactions on Software Engineering and Methodology* 30.3. DOI: 10.1145/3447808.

Chen, Ting, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang (2019). "TokenScope: Automatically Detecting Inconsistent Behaviors of Cryptocurrency Tokens in Ethereum". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, pp. 1503–1520. DOI: 10.1145/3319535.3345664. URL: https://doi.org/10.1145/3319535.3345664.

Chen, Weili, Zibin Zheng, Jiahui Cui, Edith Ngai, Peilin Zheng, and Yuren Zhou (2018a). "Detecting Ponzi Schemes on Ethereum: Towards Healthier Blockchain

Technology". In: *Proceedings of the 2018 World Wide Web Conference*. WWW '18. Lyon, France: International World Wide Web Conferences Steering Committee, pp. 1409–1418. DOI: 10.1145/3178876.3186046. URL: https://doi.org/10.1145/3178876.3186046.

Chen, Wubing, Zhiying Xu, Shuyu Shi, Yang Zhao, and Jun Zhao (Dec. 2018b). "A Survey of Blockchain Applications in Different Domains". In: *Proceedings of the 2018 International Conference on Blockchain Technology and Application*. ICBTA 2018. ACM, pp. 17–21. DOI: 10.1145/3301403.3301407. URL: https://dl.acm.org/doi/10.1145/3301403.3301407.

Chen, Xin, Dongjin Yu, Xulin Fan, Lin Wang, and Jie Chen (2022d). "Multiclass Classification for Self-Admitted Technical Debt Based on XGBoost". In: *IEEE Transactions on Reliability* 71.3, pp. 1309–1324. DOI: 10.1109/TR.2021.3087864.

Chittoda, J. (2019). *Mastering blockchain programming with Solidity : write production-ready smart contracts for Ethereum Blockchain with Solidity*. Packt Publishing. URL: https://books.google.ca/books?id=WB84yAEACAAJ.

Chu, Hanting, Pengcheng Zhang, Hai Dong, Yan Xiao, Shunhui Ji, and Wenrui Li (2023). "A survey on smart contract vulnerabilities: Data sources, detection and repair". In: *Information and Software Technology* 159, p. 107221. DOI: https://doi.org/10.1016/j.infsof.2023.107221. URL: https://www.sciencedirect.com/science/article/pii/S0950584923000757.

Cunningham, Ward (1992). "The WyCash Portfolio Management System". In: *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum)*. OOPSLA '92. Vancouver, British Columbia, Canada: Association for Computing Machinery, pp. 29–30. DOI: 10.1145/157709.157715.

DappRadar (Aug. 2021). *How dapp metrics can guide Crypto Investments*. URL: https://dappradar.com/blog/how-dapp-metrics-can-guide-crypto-investments.

de Freitas Farias, M. A., M. G. de Mendonça Neto, A. B. d. Silva, and R. O. Spínola (2015). "A Contextualized Vocabulary Model for identifying technical debt on code comments". In: *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pp. 25–32. DOI: 10.1109/MTD.2015.7332621.

Deloitte (Aug. 2023). *Using blockchain to drive supply chain transparency and Innovation*. URL: https://www2.deloitte.com/us/en/pages/operations/articles/blockchain-supply-chain-innovation.html.

Dharma (2019). *Dharma-smart-wallet: UpgradeableBeacon*. Accessed: 03-March-2025. URL: https://github.com/dharma-eng/dharma-smart-wallet.

Diligence, ConsenSys (2017). URL: https://consensys.github.io/smart-contract-best-practices/development-recommendations/precautions/upgradeability/.

Ebrahimi, Amir Mohammad, Gustavo A. Oliva, and Ahmed E. Hassan (2023). "Self-Admitted Technical Debt in Ethereum Smart Contracts: A Large-Scale Exploratory Study". In: *IEEE Transactions on Software Engineering* 49.9, pp. 4304–4323. DOI: 10.1109/TSE.2023.3289808.

Eisenhardt, Kathleen M. (1989). "Building Theories from Case Study Research". In: *The Academy of Management Review* 14.4, pp. 532–550. URL: http://www.jstor.org/stable/258557.

Entriken, William (Nov. 2022). *EIP-6049: Deprecate SELFDESTRUCT*. Accessed: 03-March-2025. URL: https://eips.ethereum.org/EIPS/eip-6049.

Ethereum (2023a). *Upgrading Smart Contracts*. Accessed: 2023-08-22.

Ethereum (Aug. 2023b). *Upgrading smart contracts*. Accessed: 03-March-2025. URL: https://ethereum.org/en/developers/docs/smart-contracts/upgrading/%5C#proxy-patterns.

Ethereum (Oct. 2025a). *Ethereum accounts*. Accessed: 03-July-2025. URL: https://ethereum.org/developers/docs/accounts/.

Ethereum (2025b). *Proof-of-stake (PoS)*. Accessed: 2025-08-22.

Ethereum ETL (2019). URL: https://ethereum-etl.readthedocs.io/en/latest/.

Etherscan (2023a). *Decentralized Applications (DApps): An Overview*. Accessed: 2025-08-22.

Etherscan (2023b). *Understanding Ethereum Accounts*. Accessed: 03-July-2025. URL: https://info.etherscan.com/understanding-ethereum-accounts/.

FasterCapital (June 2024). *Blockchain Kpis: Key performance indicators: Smart contracts: Smarter metrics: Kpis for blockchain applications*.

Feist, Josselin, Gustavo Greico, and Alex Groce (2019). "Slither: a static analysis framework for smart contracts". In: *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*. WETSEB '19. Montreal, Quebec, Canada: IEEE Press, pp. 8–15. DOI: 10.1109/WETSEB.2019.00008. URL: https://doi.org/10.1109/WETSEB.2019.00008.

Flisar, Jernej and Vili Podgorelec (2019). "Identification of Self-Admitted Technical Debt Using Enhanced Feature Selection Based on Word Embedding". In: *IEEE Access* 7, pp. 106475–106494. DOI: 10.1109/ACCESS.2019.2933318.

Fortune Business Insights (Aug. 2023). *Decentralized Finance Technology Market Size, share & covid-19 impact analysis, by component (decentralized applications (DAPPS) and smart contracts), by application (Asset Management, compliance and KYT, data analytics, payments, and gaming), by industry (BFSI, retail & e-commerce, Media & Entertainment, Automotive, and others), and Regional Forecast, 2023 – 2030*. URL: https://www.fortunebusinessinsights.com/decentralized-finance-technology-market-107823.

Fravoll (2018). *Proxy delegate*. Accessed: 03-March-2025. URL: https://fravoll.github.io/solidity-patterns/proxy_delegate.html.

Freitas Farias, Mário André de, Manoel Gomes de Mendonça Neto, Marcos Kalinowski, and Rodrigo Oliveira Spínola (2020). "Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary". In: *Information and Software Technology* 121, p. 106270. DOI: https://doi.org/10.1016/j.infsof.2020.106270. URL: https://www.sciencedirect.com/science/article/pii/S0950584920300203.

Fröwis, Michael and Rainer Böhme (2017). "In Code We Trust? Measuring the Control Flow Immutability of All Smart Contracts Deployed on Ethereum". In: *DPM/CBT@ESORICS*. URL: https://api.semanticscholar.org/CorpusID:33513899.

Fröwis, Michael and Rainer Böhme (2022). "Not All Code are Create2 Equal". In: *Financial Cryptography and Data Security. FC 2022 International Workshops: CoDecFin, DeFi, Voting, WTSC, Grenada, May 6, 2022, Revised Selected Papers*. Grenada, Grenada: Springer-Verlag, pp. 516–538. DOI: 10.1007/978-3-031-32415-4_32. URL: https://doi.org/10.1007/978-3-031-32415-4_32.

Furlonger, D. and C. Uzureau (2019). *The Real Business of Blockchain: How Leaders Can Create Value in a New Digital Age*. Harvard Business Review Press. URL: https://books.google.ca/books?id=obCPDwAAQBAJ.

Gabriel Barros, Patrick Gallagher (Mar. 2019a). *EIP-1822: Universal upgradeable proxy standard (UUPS) [draft]*. URL: https://eips.ethereum.org/EIPS/eip-1822.

Gabriel Barros, Patrick Gallagher (Mar. 2019b). *ERC-1822: Universal Upgradeable Proxy Standard (UUPS)*. Accessed: 03-March-2025. URL: https://eips.ethereum.org/EIPS/eip-1822.

Gamma, Erich, Richard Helm, Ralph Johnson, and John M. Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Longman Publishing Co., Inc.

Gao, Z., L. Jiang, X. Xia, D. Lo, and J. Grundy (2020). "Checking Smart Contracts with Structural Code Embedding". In: *IEEE Transactions on Software Engineering*. DOI: 10.1109/TSE.2020.2971482.

Gartner (Aug. 2021). *Gartner Identifies Key Emerging Technologies Spurring Innovation Through Trust, Growth and Change*. Accessed: 03-March-2025. URL: https://www.gartner.com/en/newsroom/press-releases/2021-08-23-gartner-identifies-key-emerging-technologies-spurring-innovation-through-trust-growth-and-change.

Gnosis (2023). *Gnosis/EVM-proxy-detection: Detect proxy contracts and their target addresses using ethers*. Accessed: 03-March-2025. URL: https://github.com/gnosis/evm-proxy-detection.

Google (2018). URL: https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-how-we-built-dataset.

Grech, Neville, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis (2019). "Gigahorse: Thorough, Declarative Decompilation of Smart Contracts". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 1176–1186. DOI: 10.1109/ICSE.2019.00120.

Grech, Neville, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis (Sept. 2020). "MadMax: Analyzing the out-of-Gas World of Smart Contracts". In: *Commun. ACM* 63.10, pp. 87–95. DOI: 10.1145/3416262.

Guo, Yuepu, Carolyn Seaman, Rebeka Gomes, Antonio Cavalcanti, Graziela Tonin, Fabio Q. B. Da Silva, André L. M. Santos, and Clauirton Siebra (2011). "Tracking technical debt — An exploratory case study". In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pp. 528–531. DOI: 10.1109/ICSM.2011.6080824.

Guo, Zhaoqiang, Shiran Liu, Jinping Liu, Yanhui Li, Lin Chen, Hongmin Lu, and Yuming Zhou (July 2021). "How Far Have We Progressed in Identifying Self-admitted Technical Debts? A Comprehensive Empirical Study". In: *ACM Trans. Softw. Eng. Methodol.* 30.4. DOI: 10.1145/3447247. URL: https://doi.org/10.1145/3447247.

Guo, Zhaoqiang, Shiran Liu, Jinping Liu, Yanhui Li, Lin Chen, Hongmin Lu, Yuming Zhou, and Baowen Xu (2019). *MAT: A simple yet strong baseline for identifying self-admitted technical debt*. arXiv: 1910.13238 [cs.SE]. URL: https://arxiv.org/abs/1910.13238.

Gwet, Kilem (Jan. 2002). “Inter-Rater Reliability: Dependency on Trait Prevalence and Marginal Homogeneity”. In: *Stat Methods Inter-Rater Reliab Assess* 2.

Hassan, Safwat, Weiyi Shang, and Ahmed E. Hassan (Feb. 2017). “An empirical study of emergency updates for top android mobile apps”. In: *Empirical Softw. Engg.* 22.1, pp. 505–546. DOI: 10.1007/s10664-016-9435-7. URL: https://doi.org/10.1007/s10664-016-9435-7.

Hassan, Safwat M. I. (2018). *Studying User-Developer Interactions Through the Updating and Reviewing Mechanisms of the Google Play Store*. English. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Analyte descriptor - N.A; Last updated - 2024-03-15. URL: https://sail.cs.queensu.ca/data/pdfs/Hassan_Safwat_MI_201809_PHD.pdf.

He, Ningyu, Lei Wu, Haoyu Wang, Yao Guo, and Xuxian Jiang (2019). “Characterizing Code Clones in the Ethereum Smart Contract Ecosystem”. In: *CoRR* abs/1905.00272. arXiv: 1905.00272. URL: http://arxiv.org/abs/1905.00272.

Hirao, Toshiki, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto (2019). “The Review Linkage Graph for Code Review Analytics: A Recovery Approach and Empirical Study”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, pp. 578–589. DOI: 10.1145/3338906.3338949.

Hsieh, Hsiu-Fang and Sarah E. Shannon (2005). “Three Approaches to Qualitative Content Analysis”. In: *Qualitative Health Research* 15.9. PMID: 16204405,

pp. 1277–1288. DOI: 10.1177/1049732305276687. URL: https://doi.org/10.1177/1049732305276687.

Huang, Jianjun, Songming Han, Wei You, Wenchang Shi, Bin Liang, Jingzheng Wu, and Yanjun Wu (2021). “Hunting Vulnerable Smart Contracts via Graph Embedding Based Bytecode Matching”. In: *IEEE Transactions on Information Forensics and Security* 16, pp. 2144–2156. DOI: 10.1109/TIFS.2021.3050051.

Huang, Q., E. Shihab, X. Xia, L. David, and L. Shanping (2017). “Identifying self-admitted technical debt in open source projects using text mining”. In: *Empirical Software Engineering* 23, pp. 418–451. DOI: 10.1007/s10664-017-9522-4.

Huang, Yuan, Xiaoyuan Wu, Quanqi Wang, Ziang Qian, Xiangping Chen, Mingdong Tang, and Zibin Zheng (2024). “The Sword of Damocles: Upgradeable Smart Contract in Ethereum”. In: *2024 IEEE/ACM 32nd International Conference on Program Comprehension (ICPC)*, pp. 333–345.

Jiang, Lingxiao, Ghassan Misherghi, Zhendong Su, and Stephane Glondu (2007). “DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones”. In: *Proceedings of the 29th International Conference on Software Engineering*. ICSE ’07. Washington, DC, USA: IEEE Computer Society, pp. 96–105. DOI: 10.1109/ICSE.2007.30.

Johnson, Nick (Nov. 2016). *The first upgradeable proxy contract*. URL: https://gist.github.com/Arachnid/4ca9da48d51e23e5cfe0f0e14dd6318f.

Jorge Izquierdo, Manuel Araoz (Feb. 2018). *ERC-897: DelegateProxy*. Accessed: 03-March-2025. URL: https://eips.ethereum.org/EIPS/eip-897.

Kalra, Sukrit, Seep Goel, Mohan Dhawan, and Subodh Sharma (2018). "ZEUS: Analyzing Safety of Smart Contracts". In: *Proceedings of the 25th Annual Network and Distributed System Security Symposium*. NDSS '18. URL: https://api.semanticscholar.org/CorpusID:3481056.

Kannengiesser, Niclas, Sebastian Lins, Christian Sander, Klaus Winter, Hellmuth Frey, and Ali Sunyaev (2021). "Challenges and Common Solutions in Smart Contract Development". In: *IEEE Transactions on Software Engineering*, pp. 1–1. DOI: 10.1109/TSE.2021.3116808.

Kashiwa, Yutaro, Ryoma Nishikawa, Yasutaka Kamei, Masanari Kondo, Emad Shihab, Ryosuke Sato, and Naoyasu Ubayashi (2022). "An empirical study on self-admitted technical debt in modern code review". In: *Information and Software Technology* 146, p. 106855. DOI: https://doi.org/10.1016/j.infsof.2022.106855. URL: https://www.sciencedirect.com/science/article/pii/S0950584922000258.

Kiffer, Lucianna, Dave Levin, and Alan Mislove (2018). "Analyzing Ethereum's Contract Topology". In: *Proceedings of the Internet Measurement Conference 2018*. IMC '18. Boston, MA, USA: Association for Computing Machinery, pp. 494–499. DOI: 10.1145/3278532.3278575.

Kondo, Masanari, Gustavo A. Oliva, Zhen Ming Jack Jiang, Ahmed E. Hassan, and Osamu Mizuno (Nov. 2020). "Code Cloning in Smart Contracts: A Case Study on Verified Contracts from the Ethereum Blockchain Platform". In: *Empirical Software Engineering* 25.6, pp. 4617–4675. DOI: 10.1007/s10664-020-09852-5. URL: https://doi.org/10.1007/s10664-020-09852-5.

Kruchten, Philippe, Robert Nord, and Ipek Ozkaya (Apr. 2019). *Managing Technical Debt: Reducing Friction in Software Development*. Accessed: 2025-Jun-12. Carnegie Mellon University, Software Engineering Institute's Digital Library. URL: https://insights.sei.cmu.edu/library/managing-technical-debt-reducing-friction-in-software-development/.

Labazova, Olga, Tobias Dehling, and Ali Sunyaev (2019). "From Hype to Reality: A Taxonomy of Blockchain Applications". In: *Cybersecurity*. URL: https://api.semanticscholar.org/CorpusID:108382812.

Lacity, Mary and Remko Van Hoek (Feb. 2021). *What we have learned so far about blockchain for business*.

Lehman, M.M., J.F. Ramil, P.D. Wernick, D.E. Perry, and W.M. Turski (1997). "Metrics and laws of software evolution-the nineties view". In: *Proceedings Fourth International Software Metrics Symposium*, pp. 20–32. DOI: 10.1109/METRIC.1997.637156.

Li, Ming, Qian Gao, and Tianfei Yu (Aug. 2023a). "Kappa statistic considerations in evaluating inter-rater reliability between two raters: which, when and context matters". In: *BMC Cancer* 23. DOI: 10.1186/s12885-023-11325-z.

Li, Wenkai, Xiaoqi Li, Zongwei Li, and Yuqing Zhang (2024a). "COBRA: Interaction-Aware Bytecode-Level Vulnerability Detector for Smart Contracts". In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE '24. Sacramento, CA, USA: Association for Computing Machinery, pp. 1358–1369. DOI: 10.1145/3691620.3695601. URL: https://doi.org/10.1145/3691620.3695601.

Li, Wenkai, Xiaoqi Li, Zongwei Li, and Yuqing Zhang (2024b). “COBRA: Interaction-Aware Bytecode-Level Vulnerability Detector for Smart Contracts”. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’24. Sacramento, CA, USA: Association for Computing Machinery, pp. 1358–1369. DOI: 10.1145/3691620.3695601. URL: https://doi.org/10.1145/3691620.3695601.

Li, Xiaofan, Jin Yang, Jiaqi Chen, Yuzhe Tang, and Xing Gao (2024c). *Characterizing Ethereum Upgradable Smart Contracts and Their Security Implications*. arXiv: 2403.01290 [cs.CR]. URL: https://arxiv.org/abs/2403.01290.

Li, Xiaoqi, Ting Chen, Xiapu Luo, Tao Zhang, Le Yu, and Zhou Xu (2020). “STAN: Towards Describing Bytecodes of Smart Contract”. In: *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, pp. 273–284. DOI: 10.1109/QRS51102.2020.00045.

Li, Yikun, Mohamed Soliman, Paris Avgeriou, and Lou Somers (2023b). “Self-Admitted Technical Debt in the Embedded Systems Industry: An Exploratory Case Study”. In: *IEEE Transactions on Software Engineering* 49.4, pp. 2545–2565. DOI: 10.1109/TSE.2022.3224378.

Lin, Dayi, Cor-Paul Bezemer, and Ahmed E. Hassan (Dec. 2019a). “Identifying gameplay videos that exhibit bugs in computer games”. In: *Empirical Software Engineering*. DOI: 10.1007/s10664-019-09733-6.

Lin, Dayi, Cor-Paul Bezemer, Ying Zou, and Ahmed E. Hassan (Feb. 2019b). “An Empirical Study of Game Reviews on the Steam Platform”. In: *Empirical Software Engineering* 24. DOI: 10.1007/s10664-018-9627-4.

Lin, Jiahuei, Haoxiang Zhang, Bram Adams, and Ahmed E Hassan (2023). "Vulnerability management in Linux distributions: An empirical study on Debian and Fedora". In: *Empirical Software Engineering* 28.2, p. 47.

Liu, Jiakun, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li (2020). "Is Using Deep Learning Frameworks Free? Characterizing Technical Debt in Deep Learning Frameworks". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society*. ICSE-SEIS '20. Seoul, South Korea: Association for Computing Machinery, pp. 1–10. DOI: 10.1145/3377815.3381377.

Liu, Xia, Baojian Hua, Yang Wang, and Zhizhong Pan (2023a). "An Empirical Study of Smart Contract Decompilers". In: *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 1–12. DOI: 10.1109/SANER56733.2023.00011.

Liu, Yue, Qinghua Lu, Liming Zhu, Hye-Young Paik, and Mark Staples (2023b). "A systematic literature review on blockchain governance". In: *Journal of Systems and Software* 197, p. 111576. DOI: https://doi.org/10.1016/j.jss.2022.111576. URL: https://www.sciencedirect.com/science/article/pii/S0164121222002527.

Liu, Zhongxin, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li (2018). "SATD detector: a text-mining-based self-admitted technical debt detection tool". In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, pp. 9–12. DOI: 10.1145/3183440.3183478. URL: https://doi.org/10.1145/3183440.3183478.

Lohr, Matthias and Sven Peldszus (2020). “Maintenance of Long-Living Smart Contracts”. In: *Software Engineering*. URL: https://api.semanticscholar.org/CorpusID:214625910.

Loo, Andrew (2025). *What are dApps?* Accessed June 30, 2025. Corporate Finance Institute. URL: https://corporatefinanceinstitute.com/resources/cryptocurrency/what-are-dapps/.

Luu, Loi, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor (2016). “Making Smart Contracts Smarter”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: Association for Computing Machinery, pp. 254–269. DOI: 10.1145/2976749.2978309. URL: https://doi.org/10.1145/2976749.2978309.

M., Girish N. and Sharadadevi Kaganurmath (July 2022). “Upgradability of smart contracts: A Review”. In: *International Research Journal of Engineering and Technology (IRJET)* 9.7, pp. 2602–2605. URL: https://www.irjet.net/archives/V9/i7/IRJET-V9I722.

Maipradit, Rungroj, Bin Lin, Csaba Nagy, Gabriele Bavota, Michele Lanza, Hideaki Hata, and Kenichi Matsumoto (2020). “Automated Identification of On-hold Self-admitted Technical Debt”. In: *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 54–64. DOI: 10.1109/SCAM51674.2020.00011.

Maldonado, E., E. Shihab, and N. Tsantalis (2017a). “Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt”. In: *IEEE Transactions on Software Engineering* 43.11, pp. 1044–1062. DOI: 10.1109/TSE.2017.2654244.

Maldonado, Everton Da S., Rabe Abdalkareem, Emad Shihab, and Alexander Serebrenik (2017b). "An Empirical Study on the Removal of Self-Admitted Technical Debt". In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 238–248. DOI: 10.1109/ICSME.2017.8.

Marx, Steve (2018). URL: https://programtheblockchain.com/posts/2018/07/25/anatomy-of-transaction-data.

McHugh, Mary (Oct. 2012). "Interrater reliability: The kappa statistic". In: *Biochemia medica : časopis Hrvatskoga društva medicinskih biokemičara / HDMB* 22, pp. 276–82. DOI: 10.11613/BM.2012.031.

Mcnemar, Quinn (1947). "Note on the sampling error of the difference between correlated proportions or percentages". In: *Psychometrika* 12, pp. 153–157. URL: https://api.semanticscholar.org/CorpusID:46226024.

Mehar, Izhar, Charles Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry Kim, and Marek Laskowski (Jan. 2019). "Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack". In: *Journal of Cases on Information Technology* 21, pp. 19–32. DOI: 10.4018/JCIT.2019010102.

Meisami, Sajad and William Edward Bodell III (2023). *A Comprehensive Survey of Upgradeable Smart Contract Patterns*. arXiv: 2304.03405 [cs.CR]. URL: https://arxiv.org/abs/2304.03405.

Miraz, Mahdi H. and Maaruf Ali (Jan. 2018). "Applications of Blockchain Technology beyond Cryptocurrency". In: *Annals of Emerging Technologies in Computing* 2.1, pp. 1–6. DOI: 10.33166/aetic.2018.01.001. URL: http://dx.doi.org/10.33166/AETiC.2018.01.001.

Mudge, Nick (Oct. 2018a). *ERC-1538: Transparent Contract Standard*. Accessed: 03-March-2025. URL: https://eips.ethereum.org/EIPS/eip-1538.

Mudge, Nick (Oct. 2018b). *ERC-1538: Transparent contract standard [draft]*. URL: https://eips.ethereum.org/EIPS/eip-1538.

Mudge, Nick (Feb. 2020a). *ERC-2535: Diamonds, Multi-Facet Proxy*. Accessed: 03-March-2025. URL: https://eips.ethereum.org/EIPS/eip-2535.

Mudge, Nick (Feb. 2020b). *ERC-2535: Diamonds, multi-facet proxy*. URL: https://eips.ethereum.org/EIPS/eip-2535.

Mudge, Nick (2020c). *Three Diamond Reference Implementations*. URL: https://github.com/mudgen/diamond.

Mudge, Nick (2022). *Mudgen/Awesome-Diamonds: A curated list of awesome EIP2535 Diamonds Resources, libraries, tools, articles and more*. Accessed: 03-March-2025. URL: https://github.com/mudgen/awesome-diamonds.

Nanjundeswaraswamy, Dr and Shilpa Divakara (Mar. 2021). "DETERMINATION OF SAMPLE SIZE AND SAMPLING METHODS IN APPLIED RESEARCH". In: *Proceedings on Engineering Sciences* 3, pp. 25–32. DOI: 10.24874/PES03.01.003. URL: https://api.semanticscholar.org/CorpusID:233708517.

Nanjundeswaraswamy, TS and Shilpa Divakar (2021). "Determination of sample size and sampling methods in applied research". In: *Proceedings on engineering sciences* 3.1, pp. 25–32.

Ngaogate, Wasana (May 2019). "GoF Design Patterns in a Smart City System". In: *Journal of Software* 14, pp. 220–226. DOI: 10.17706/jsw.14.5.220-226.

Norvill, Robert, Beltran Borja Fiz Pontiveros, Radu State, Irfan Awan, and Andrea Cullen (2017). "Automated Labeling of Unknown Contracts in Ethereum". In:

2017 26th International Conference on Computer Communication and Networks (ICCCN), pp. 1–6. DOI: 10.1109/ICCCN.2017.8038513.

Nugroho, Ariadi, Joost Visser, and Tobias Kuipers (2011). "An Empirical Model of Technical Debt and Interest". In: *Proceedings of the 2nd Workshop on Managing Technical Debt*. MTD '11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, pp. 1–8. DOI: 10.1145/1985362.1985364.

OBrien, David, Sumon Biswas, Sayem Imtiaz, Rabe Abdalkareem, Emad Shihab, and Hridesh Rajan (2022). "23 shades of self-admitted technical debt: an empirical study on machine learning software". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, pp. 734–746. DOI: 10.1145/3540250.3549088. URL: https://doi.org/10.1145/3540250.3549088.

Oliva, Gustavo A. and Ahmed E. Hassan (2021). "The gas triangle and its challenges to the development of blockchain-powered applications". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, pp. 1463–1466. DOI: 10.1145/3468264.3473130. URL: https://doi.org/10.1145/3468264.3473130.

Oliva, Gustavo A., Ahmed E. Hassan, and Zhen Ming (Jack) Jiang (2020). "An exploratory study of smart contracts in the Ethereum blockchain platform". In: *Empirical Software Engineering* 25 (3).

OpenZeppelin (2017). *Proxies*. Accessed: 03-March-2025. URL: https://docs.openzeppelin.com/contracts/4.x/api/proxy#Proxy.

OpenZeppelin (Aug. 2018a). *Proxy patterns*. Accessed: 03-March-2025. URL: https://blog.openzeppelin.com/proxy-patterns/.

OpenZeppelin (2018b). *Proxy upgrade pattern*. URL: https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies.

OpenZeppelin (2018c). *Upgradeability using Eternal Storage*. URL: https://github.com/OpenZeppelin/openzeppelin-labs/tree/master/upgradeability_using_eternal_storage.

OpenZeppelin (2018d). *Upgradeability using Inherited Storage*. URL: https://github.com/OpenZeppelin/openzeppelin-labs/tree/master/upgradeability_using_inherited_storage.

OpenZeppelin (2018e). *Upgradeability using unstructured storage*. URL: https://github.com/OpenZeppelin/openzeppelin-labs/tree/master/upgradeability_using_unstructured_storage.

OpenZeppelin (2018f). *Writing upgradeable contracts*. URL: https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable.

OpenZeppelin (2023a). *How to set up on-chain governance*. Accessed: 03-March-2025. URL: https://docs.openzeppelin.com/contracts/4.x/governance.

OpenZeppelin (2023b). *Using with upgrades*. Accessed: 03-March-2025. URL: https://docs.openzeppelin.com/contracts/4.x/upgradeable.

Pacheco, Michael, Gustavo Oliva, Gopi Krishnan Rajbahadur, and Ahmed Hassan (Apr. 2023a). "Is My Transaction Done Yet? An Empirical Study of Transaction Processing Times in the Ethereum Blockchain Platform". In: *ACM Trans. Softw. Eng. Methodol.* 32.3. DOI: 10.1145/3549542. URL: https://doi.org/10.1145/3549542.

Pacheco, Michael, Gustavo A. Oliva, Gopi Krishnan Rajbahadur, and Ahmed E. Hassan (2022). *What makes Ethereum blockchain transactions be processed fast or slow? An empirical study*. arXiv: 2206.08905 [cs.SE].

Pacheco, Michael, Gustavo A. Oliva, Gopi Krishnan Rajbahadur, and Ahmed E. Hassan (Feb. 2023b). "What makes Ethereum blockchain transactions be processed fast or slow? An empirical study". In: *Empirical Software Engineering* 28.2, p. 39. DOI: 10.1007/s10664-022-10283-7. URL: https://doi.org/10.1007/s10664-022-10283-7.

Palkeo (2019). *Panoramix: Ethereum Decompiler*. URL: https://github.com/palkeo/panoramix.

Palladino, Patricio (2018). *Malicious backdoors in Ethereum Proxies*. Accessed: 03-March-2025. URL: https://medium.com/nomic-foundation-blog/malicious-backdoors-in-ethereum-proxies-62629adf3357.

Palladino, Santiago (Oct. 2020). *The State of Smart Contract Upgrades*. Accessed: 03-March-2025. URL: https://blog.openzeppelin.com/the-state-of-smart-contract-upgrades/.

Palladino, Santiago (Apr. 2021). *The transparent proxy pattern*. URL: https://blog.openzeppelin.com/the-transparent-proxy-pattern/.

Pathade, Chetan and Shweta Hooli (2025). *Exposing Hidden Backdoors in NFT Smart Contracts: A Static Security Analysis of Rug Pull Patterns*. arXiv: 2506.07974 [cs.CR]. URL: https://arxiv.org/abs/2506.07974.

Patra, Ram (2024). *Understanding call, delegatecall, and staticcall in solidity with real-world analogy*. URL: https://blog.rampatra.com/understanding-call-delegatecall-and-staticcall-in-solidity-with-real-world-analogy.

Peh, Bernard (May 2018). *The basics of upgradable proxy contracts in Ethereum*. URL: https://medium.com/@blockchain101/the-basics-of-upgradable-proxy-contracts-in-ethereum-479b5d3363d6.

Peter Murray Nate Welch, Joe Messerman (June 2018a). *ERC-1167: Minimal Proxy Contract*. Accessed: 03-March-2025. URL: https://eips.ethereum.org/EIPS/eip-1167.

Peter Murray, Nate Welch (June 2018b). *EIP-1167: Minimal proxy contract*. URL: https://eips.ethereum.org/EIPS/eip-1167.

pinkiebell (Mar. 2021). *ERC-3448: MetaProxy Standard*. Accessed: 03-March-2025. URL: https://eips.ethereum.org/EIPS/eip-3448.

Pinna, Andrea, Maria Ilaria Lunesu, Stefano Orrù, and Roberto Tonelli (2023). "Investigation on Self-Admitted Technical Debt in Open-Source Blockchain Projects". In: *Future Internet* 15.7. DOI: 10.3390/fi15070232. URL: https://www.mdpi.com/1999-5903/15/7/232.

Porru, Simone, Andrea Pinna, Michele Marchesi, and Roberto Tonelli (2017). "Blockchain-Oriented Software Engineering: Challenges and New Directions". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 169–171. DOI: 10.1109/ICSE-C.2017.142.

Porter, Sam (May 2021). *ERC-3561: Trust Minimized Upgradeability Proxy*. Accessed: 03-March-2025. URL: https://eips.ethereum.org/EIPS/eip-3561.

Potdar, A. and E. Shihab (2014). "An Exploratory Study on Self-Admitted Technical Debt". In: *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 91–100. DOI: 10.1109/ICSME.2014.31.

Prenner, Julian Aron and Romain Robbes (2021). *Making the most of small Software Engineering datasets with modern machine learning*. arXiv: 2106.15209 [cs.SE]. URL: https://arxiv.org/abs/2106.15209.

Qasse, Ilham, Mohammad Hamdaqa, and Björn Þór Jónsson (2023). *Smart Contract Upgradeability on the Ethereum Blockchain Platform: An Exploratory Study*. arXiv: 2304.06568 [cs.SE].

Qasse, Ilham, Mohammad Hamdaqa, and Björn Þór Jónsson (2024). *Immutable in Principle, Upgradeable by Design: Exploratory Study of Smart Contract Upgradeability*. arXiv: 2407.01493 [cs.SE]. URL: https://arxiv.org/abs/2407.01493.

Qasse, Ilham Ahmed, Josef Spillner, Manar Abu Talib, and Qassim Nasir (2020). "A Study on ĐApps Characteristics". In: *2020 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, pp. 88–93. DOI: 10.1109/DAPPS49028.2020.00010.

Qu, Yubin, Tie Bao, Xiang Chen, Long Li, Xianzhen Dou, Meng Yuan, and Hongmei Wang and (2022a). "Do we need to pay technical debt in blockchain software systems?" In: *Connection Science* 34.1, pp. 2026–2047. DOI: 10.1080/09540091.2022.2067125. eprint: https://doi.org/10.1080/09540091.2022.2067125. URL: https://doi.org/10.1080/09540091.2022.2067125.

Qu, Yubin, W. Eric Wong, and Dongcheng Li (2022b). "Empirical Research for Self-Admitted Technical Debt Detection in Blockchain Software Projects". In: *International Journal of Performability Engineering* 18.3, 149, p. 149. DOI: 10.23940/ijpe.22.03.p1.149157. URL: https://www.ijpe-online.com/EN/abstract/article_4664.shtml.

Ralph, Paul, Nauman bin Ali, Sebastian Baltes, Domenico Bianculli, Jessica Diaz, Yvonne Dittrich, Neil Ernst, Michael Felderer, Robert Feldt, Antonio Filieri, Breno Bernard Nicolau de França, Carlo Alberto Furia, Greg Gay, Nicolas Gold, Daniel Graziotin, Pinjia He, Rashina Hoda, Natalia Juristo, Barbara Kitchenham, Valentina Lenarduzzi, Jorge Martínez, Jorge Melegati, Daniel Mendez, Tim Menzies, Jefferson Molleri, Dietmar Pfahl, Romain Robbes, Daniel Russo, Nyyti Saarimäki, Federica Sarro, Davide Taibi, Janet Siegmund, Diomidis Spinellis, Miroslaw Staron, Klaas Stol, Margaret-Anne Storey, Davide Taibi, Damian Tamburri, Marco Torchiano, Christoph Treude, Burak Turhan, Xiaofeng Wang, and Sira Vegas (2021). *Empirical Standards for Software Engineering Research*. arXiv: 2010.03525 [cs.SE]. URL: https://arxiv.org/abs/2010.03525.

Ren, Xiaoxue, Zhenchang Xing, Xin Xia, David Lo, Xinyu Wang, and John Grundy (July 2019). "Neural Network-Based Detection of Self-Admitted Technical Debt: From Performance to Explainability". In: *ACM Trans. Softw. Eng. Methodol.* 28.3. DOI: 10.1145/3324916.

Richardson, C. (2018). *Microservices Patterns: With examples in Java*. Manning. URL: https://books.google.ca/books?id=UeK1swEACAAJ.

Rios, Nicolli, Manoel Gomes de Mendonça Neto, and Rodrigo Oliveira Spínola (2018). "A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners". In: *Information and Software Technology* 102, pp. 117–145. DOI: https://doi.org/10.1016/j.infsof.2018.05.010.

Romano, J., J.D. Kromrey, J. Coraggio, and J. Skowronek (2006). "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for

evaluating group differences on the NSSE and other surveys?" In: *annual meeting of the Florida Association of Institutional Research*, pp. 1–3. URL: https://www.bibsonomy.org/bibtex/216a5c27e770147e5796719fc6b68547d/kweiand.

Rosa, Giovanni, Simone Scalabrino, Stefano Mastrostefano, and Rocco Oliveto (Mar. 2025). "Why and how developers maintain smart contracts". In: *Empirical Softw. Engg.* 30.3. DOI: 10.1007/s10664-025-10639-9. URL: https://doi.org/10.1007/s10664-025-10639-9.

Ruaro, Nicola, Fabio Gritti, Robert McLaughlin, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna (Jan. 2024). "Not your Type! Detecting Storage Collision Vulnerabilities in Ethereum Smart Contracts". In: DOI: 10.14722/ndss.2024.24713.

S. Maldonado, E. d. and E. Shihab (2015). "Detecting and quantifying different types of self-admitted technical Debt". In: *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pp. 9–15. DOI: 10.1109/MTD.2015.7332619.

Sala, Irene, Antonela Tommasel, and Francesca Arcelli Fontana (2021). "DebtHunter: A Machine Learning-based Approach for Detecting Self-Admitted Technical Debt". In: *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*. EASE '21. Trondheim, Norway: Association for Computing Machinery, pp. 278–283. DOI: 10.1145/3463274.3464455. URL: https://doi.org/10.1145/3463274.3464455.

Salehi, Mehdi, Jeremy Clark, and Mohammad Mannan (2022). *Not so immutable: Upgradeability of Smart Contracts on Ethereum*. DOI: 10.48550/ARXIV.2206.00716. URL: https://arxiv.org/abs/2206.00716.

Santiago Palladino, Francisco Giordano (Apr. 2019a). *EIP-1967: Proxy storage slots*. URL: https://eips.ethereum.org/EIPS/eip-1967.

Santiago Palladino Francisco Giordano, Hadrien Croubois (Apr. 2019b). *ERC-1967: Proxy Storage Slots*. Accessed: 03-March-2025. URL: https://eips.ethereum.org/EIPS/eip-1967.

Santos, Rafael, Israel Santos, Methanias Rodrigues Júnior, and Manoel Neto (2020). "Long Term-short Memory Neural Networks and Word2vec for Self-admitted Technical Debt Detection". In: *Proceedings of the 22nd International Conference on Enterprise Information Systems - Volume 2: ICEIS,* INSTICC. SciTePress, pp. 157–165. DOI: 10.5220/0009796001570165.

Shapiro, Marc (1986). "Structure and Encapsulation in Distributed Systems: The Proxy Principle". In: *IEEE International Conference on Distributed Computing Systems*. URL: https://api.semanticscholar.org/CorpusID:14373941.

Sheikhaei, Mohammad Sadegh, Yuan Tian, Shaowei Wang, and Bowen Xu (2024). *An Empirical Study on the Effectiveness of Large Language Models for SATD Identification and Classification*. arXiv: 2405.06806 [cs.SE]. URL: https://arxiv.org/abs/2405.06806.

Shogo Ochiai, Kai Hiroi (Oct. 2023). *ERC-7546: Upgradeable Clone for Scalable Contracts*. Accessed: 03-March-2025. URL: https://eips.ethereum.org/EIPS/eip-7546.

Sierra, Giancarlo, Emad Shihab, and Yasutaka Kamei (Feb. 2019). "A Survey of Self-Admitted Technical Debt". In: *Journal of Systems and Software* 152. DOI: 10.1016/j.jss.2019.02.056.

So, Sunbeom and Hakjoo Oh (2023). “SmartFix: Fixing Vulnerable Smart Contracts by Accelerating Generate-and-Verify Repair using Statistical Models”. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. San Francisco, CA, USA: Association for Computing Machinery, pp. 185–197. DOI: 10.1145/3611643.3616341. URL: https://doi.org/10.1145/3611643.3616341.

solidity (2023). *Introduction to Smart Contracts*. Accessed: 2023-08-22. URL: https://docs.soliditylang.org/en/v0.8.21/introduction-to-smart-contracts.html.

Sridharan, Murali, Leevi Rantala, and Mika Mäntylä (2023). “PENTACET data - 23 Million Contextual Code Comments and 250,000 SATD comments”. In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pp. 412–416. DOI: 10.1109/MSR59073.2023.00063.

Suiche, Matt (2017). *The $280M Ethereum’s Parity Bug*. Accessed: 03-March-2025. URL: https://medium.com/comae/the-280m-ethereums-bug-f28e5de43513.

Sutoyo, Edi and Andrea Capiluppi (2024). *Self-Admitted Technical Debt Detection Approaches: A Decade Systematic Review*. arXiv: 2312.15020 [cs.SE]. URL: https://arxiv.org/abs/2312.15020.

Swan, Melanie (2015). *Blockchain: Blueprint for a New Economy*. 1st. O’Reilly Media, Inc. DOI: 10.5555/3006358. URL: https://dl.acm.org/doi/10.5555/3006358.

Szabo, Nick (1994). *Smart Contracts*. Accessed: 03-March-2025. URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html.

Tagra, Ankur, Haoxiang Zhang, Gopi Krishnan Rajbahadur, and Ahmed E Hassan (2022). "Revisiting reopened bugs in open source software systems". In: *Empirical Software Engineering* 27.4, p. 92.

Tikhomirov, Sergei, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov (2018). "SmartCheck: static analysis of ethereum smart contracts". In: *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. WETSEB '18. Gothenburg, Sweden: Association for Computing Machinery, pp. 9–16. DOI: 10.1145/3194113.3194115. URL: https://doi.org/10.1145/3194113.3194115.

Tripathi, Gautami, Mohd Abdul Ahad, and Gabriella Casalino (2023). "A comprehensive review of blockchain technology: Underlying principles and historical background with future challenges". In: *Decision Analytics Journal* 9, p. 100344. DOI: https://doi.org/10.1016/j.dajour.2023.100344. URL: https://www.sciencedirect.com/science/article/pii/S2772662223001844.

Vidoni, M. (2021). "Self-Admitted Technical Debt in R Packages: An Exploratory Study". In: *2021 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) (MSR)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 179–189. DOI: 10.1109/MSR52588.2021.00030. URL: https://doi.ieeecomputersociety.org/10.1109/MSR52588.2021.00030.

Wang, Xin, Jin Liu, Li Li, Xiao Chen, Xiao Liu, and Hao Wu (2021). "Detecting and explaining self-admitted technical debts with attention-based neural networks". In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ASE '20. Virtual Event, Australia: Association for Computing Machinery, pp. 871–882. DOI: 10.1145/3324884.3416583. URL: https://doi.org/10.1145/3324884.3416583.

Wasnik, Ankit (May 2024). *Implementing upgradeable smart contracts using proxy patterns*. URL: https://www.talentica.com/blogs/implementing-upgradeable-smart-contracts-using-proxy-patterns/.

Wattanakriengkrai, S., N. Srisermphoak, S. Sintoplertchaikul, M. Choetkiertikul, C. Ragkhitwetsagul, T. Sunetnanta, H. Hata, and K. Matsumoto (2019). "Automatic Classifying Self-Admitted Technical Debt Using N-Gram IDF". In: *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 316–322. DOI: 10.1109/APSEC48747.2019.00050.

Wikipedia.org (Aug. 2014). *Solidity*. URL: https://en.wikipedia.org/wiki/Solidity.

Wöhrer, Maximilian and Uwe Zdun (2018). "Design Patterns for Smart Contracts in the Ethereum Ecosystem". In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 1513–1520. DOI: 10.1109/Cybermatics_2018.2018.00255.

Wöhrer, Maximilian and Uwe Zdun (2021). "DevOps for Ethereum Blockchain Smart Contracts". In: *2021 IEEE International Conference on Blockchain (Blockchain)*, pp. 244–251. DOI: 10.1109/Blockchain53845.2021.00040.

Wood, Gavin (2014). “Ethereum: A Secure Decentralised Generalised Transaction Ledger”. In: *Project Yellow Paper*.

Wood, John (June 2023). *Is there a way to avoid using a factory with the Clone pattern? Do end users care either way?* Accessed: 03-March-2025. URL: https://forum.openzeppelin.com/t/is-there-a-way-to-avoid-using-a-factory-with-the-clone-pattern-do-end-users-care-either-way/36564/6.

Xiang, Jianhang, Zhipeng Gao, Lingfeng Bao, Xing Hu, Jiayuan Chen, and Xin Xia (Feb. 2025). “Automating Comment Generation for Smart Contract from Bytecode”. In: *ACM Trans. Softw. Eng. Methodol.* 34.3. DOI: 10.1145/3699597. URL: https://doi.org/10.1145/3699597.

Xiao, Tao, Dong Wang, Shane McIntosh, Hideaki Hata, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto (2022). “Characterizing and Mitigating Self-Admitted Technical Debt in Build Systems”. In: *IEEE Transactions on Software Engineering* 48.10, pp. 4214–4228. DOI: 10.1109/TSE.2021.3115772.

Xu, Junzhou, Fan Dang, Xuan Ding, and Min Zhou (2020). “A Survey on Vulnerability Detection Tools of Smart Contract Bytecode”. In: *2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE)*, pp. 94–98. DOI: 10.1109/ICISCAE51034.2020.9236931.

Yan, Meng, Xin Xia, Emad Shihab, David Lo, Jianwei Yin, and Xiaohu Yang (2019). “Automating Change-Level Self-Admitted Technical Debt Determination”. In: *IEEE Transactions on Software Engineering* 45.12, pp. 1211–1229. DOI: 10.1109/TSE.2018.2831232.

Yang, Aidan, Safwat Hassan, Ying Zou, and Ahmed E. Hassan (Mar. 2022a). “An empirical study on release notes patterns of popular apps in the Google Play Store”. In: *Empirical Software Engineering* 27. DOI: 10.1007/s10664-021-10086-2.

Yang, Aidan Z. H., Safwat Hassan, Ying Zou, and Ahmed E. Hassan (Mar. 2022b). “An empirical study on release notes patterns of popular apps in the Google Play Store”. In: *Empirical Softw. Engg.* 27.2. DOI: 10.1007/s10664-021-10086-2. URL: https://doi.org/10.1007/s10664-021-10086-2.

Yu, Dongjin, Lin Wang, Xin Chen, and Jie Chen (Aug. 2021). “Using BiLSTM with attention mechanism to automatically detect self-admitted technical debt”. In: *Frontiers of Computer Science* 15. DOI: 10.1007/s11704-020-9281-z.

Yu, Zhe, Fahmid Morshed Fahid, Huy Tu, and Tim Menzies (2020). “Identifying Self-Admitted Technical Debts with Jitterbug: A Two-Step Approach”. In: *IEEE Transactions on Software Engineering*, pp. 1–1. DOI: 10.1109/tse.2020.3031401.

Zampetti, Fiorella, Gianmarco Fucci, Alexander Serebrenik, and Massimiliano Di Penta (Nov. 2021). “Self-admitted technical debt practices: a comparison between industry and open-source”. In: *Empirical Softw. Engg.* 26.6. DOI: 10.1007/s10664-021-10031-3. URL: https://doi.org/10.1007/s10664-021-10031-3.

Zarir, Abdullah A., Gustavo A. Oliva, Zhen M. (Jack) Jiang, and Ahmed E. Hassan (Mar. 2021). “Developing Cost-Effective Blockchain-Powered Applications: A Case Study of the Gas Usage of Smart Contract Transactions in the Ethereum Blockchain Platform”. In: *ACM Trans. Softw. Eng. Methodol.* 30.3. DOI: 10.1145/3431726. URL: https://doi.org/10.1145/3431726.

Zhang, Mengya, Preksha Shukla, Wuqi Zhang, Zhuo Zhang, Pranav Agrawal, Zhiqiang Lin, Xiangyu Zhang, and Xiaokuan Zhang (May 2025a). “An Empirical Study of

Proxy Contracts at the Ethereum Ecosystem Scale". In: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 2996–3008. DOI: 10.1109/ICSE55347.2025.00083. URL: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00083.

Zhang, Mengya, Preksha Shukla, Wuqi Zhang, Zhuo Zhang, Pranav Agrawal, Zhiqiang Lin, Xiangyu Zhang, and Xiaokuan Zhang (May 2025b). " An Empirical Study of Proxy Smart Contracts at the Ethereum Ecosystem Scale ". In: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 620–620. DOI: 10.1109/ICSE55347.2025.00083. URL: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00083.

Zheng, Zibin, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran (Apr. 2020). "An overview on smart contracts: Challenges, advances and platforms". In: *Future Generation Computer Systems* 105, pp. 475–491. DOI: 10.1016/j.future.2019.12.019. URL: http://dx.doi.org/10.1016/j.future.2019.12.019.

Zhou, Jiayuan, Shaowei Wang, Cor-Paul Bezemer, and Ahmed E. Hassan (Jan. 2020a). "Bounties on technical Q&A sites: a case study of Stack Overflow bounties". In: *Empirical Softw. Engg.* 25.1, pp. 139–177. DOI: 10.1007/s10664-019-09744-3. URL: https://doi.org/10.1007/s10664-019-09744-3.

Zhou, Jiayuan, Shaowei Wang, Cor-Paul Bezemer, and Ahmed E. Hassan (Jan. 2020b). "Bounties on technical Q&A sites: a case study of Stack Overflow bounties". In:

Empirical Softw. Engg. 25.1, pp. 139–177. DOI: 10.1007/s10664-019-09744-3. URL: https://doi.org/10.1007/s10664-019-09744-3.

Zhou, Jiayuan, Shaowei Wang, Haoxiang Zhang, Tse-Hsun Peter Chen, and Ahmed E. Hassan (July 2021). "Studying backers and hunters in bounty issue addressing process of open source projects". In: *Empirical Software Engineering* 26. DOI: 10.1007/s10664-021-09979-z.

Zhou, Yi, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey (Aug. 2018). "Erays: Reverse Engineering Ethereum's Opaque Smart Contracts". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, pp. 1371–1385. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/zhou.

Zou, Weiqin, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu (2021). "Smart Contract Development: Challenges and Opportunities". In: *IEEE Transactions on Software Engineering* 47.10, pp. 2084–2106. DOI: 10.1109/TSE.2019.2942301.

Żuk, Igor (Feb. 2024). *ERC-7613: Puppet Proxy Contract*. Accessed: 03-March-2025. URL: https://eips.ethereum.org/EIPS/eip-7613.

APPENDIX A

Appendix

A.1 Evaluation of the Employed SATD Detector Tool

To evaluate the performance of our employed SATD detector tool, we evaluate its false-positive (FP) and false-negative (FN) rates. To calculate the false-positive rate, we leverage the manual analysis conducted as part of RQ3 (Section 4.6). As part of that analysis, we investigated a statistically representative sample of 287 SATD comments and counted the number of code comments that were incorrectly classified as SATD by Jitterbug. We identified 40 (FP = 13.9%) false-positive cases. However, 42.5% of those were classified by us as false positives because the debt

mentioned in the comment had already been paid off (i.e., outdated SATD comments). It is worth mentioning that none of the existing SATD detection tools can distinguish between paid and non-paid SATD.

Furthermore, we assess the false-negative (FN) rate of Jitterbug's easy-to-find detector tool. First, we drew a statistically representative sample of code comments (95% confidence level with a confidence interval of +-5) from the set of textually-unique comments that are classified as non-SATD by Jitterbug. We obtained 383 code comments. Next, the first two researchers manually studied the sampled comments and classified them as either SATD or non-SATD. We only identified 2 (FN = 0.5%) code comments that were actual SATD instances. Such a result suggests that Jitterbug has a very low FN rate.

A.2 Practical Applications of the Proxy Pattern in DApps

Approach. To gain a more holistic understanding of proxy pattern usage, we examined its applications in the context of DApps. Specifically, we analyzed a dataset of 3,767 Ethereum DApps collected from three well-known marketplaces: `stateofthedapps.com`, `dappradar.com`, and `dapp.com`. We cross-referenced the contract addresses of these DApps with our proxy contract dataset (Section 5.5) and identified 307 DApps (8.14%) with at least one proxy contract.

We ranked these 307 DApps by activity level in ascending order. To determine a DApp's activity level, we computed the average number of inbound transactions per day across all its contracts from its "birthday" (the deployment date of its first

contract) until September 5, 2022. We refer to the total number of inbound transactions during this period as "inbound traffic", and define activity level as the ratio of inbound traffic to the DApp's age (in days).

To understand how proxies are used in practice, we selected the top 50 most active DApps and manually analyzed their 2,045 proxy contracts. The minimum, median, maximum, and standard deviation of proxy counts per DApp were 1, 2, 1,740, and 40.9, respectively. For each DApp, we clustered proxy contracts based on bytecode similarity and randomly sampled one proxy from each cluster for analysis.

Two researchers collaboratively examined the source code of these sampled proxies using a hybrid coding approach to identify their primary purpose. Based on the ERC-897 standard, proxies are classified as either Upgradeable Proxy Contracts (UPCs), which support contract evolution, or Forwarder Proxy Contracts (FPCs), which are used for purposes such as cost-effective cloning, logging, access control, and routing. If the proxy facilitated contract evolution, we labeled it as a UPC. For FPCs, we examined the implementation details to determine the specific rationale. When the observed purpose did not match existing labels in our codebook, we created and documented a new category.

Findings. *Out of the 50 DApps studied, 43 (86%) used proxy contracts for smart contract evolution, while the remaining 7 (14%) used proxies functioning as FPCs. Among the latter, six DApps adopted proxies to reduce deployment costs. A notable example is Uniswap-V1, which deployed 1,740 minimal proxies. Each proxy delegates calls to a single logic contract (i.e.,* `Vyper_contract`*), enabling code reuse without incurring repeated deployment costs. One DApp, Swerve Finance, employed two FPCs to*

extend functionality by providing getter interfaces for reading logic contract state variables. Additionally, two DApps (Element and Ether Legend) used FPCs as routing hubs, directing user requests to appropriate logic contracts, enchanting modular design.

A.3 Cost-Efficient Deployment with Proxy Contracts: A Case Study of Uniswap-V1 DApp

Approach. To demonstrate how proxy patterns can reduce deployment costs, we analyzed Uniswap-V1 (Appendix A.2). We compared two scenarios: the actual deployment (Scenario A) and a hypothetical deployment (Scenario B). In Scenario A, a single logic contract, `Vyper_contract`, is deployed once, and 1,740 minimal proxy contracts are deployed to delegate calls to it. In Scenario B, the same logic contract is independently deployed 1,740 times, creating separate full copies including logic and state.

We calculated total deployment costs for both scenarios in ETH. Deployment fees are determined by multiplying the number of gas units used by the gas price at the time of deployment. Gas prices can vary due to network congestion, miner policies, or Ethereum upgrades. For Scenario A, we scraped gas usage and fee data from Etherscan for each of the 1,740 minimal proxy deployments, and added the deployment fee of the `Vyper_contract` (0.04 ETH). For Scenario B, we used the gas units required to deploy a single instance of the `Vyper_contract` and multiplied it by the gas prices observed in Scenario A, then summed the result across all instances.

Findings. *The total gas cost for Scenario A (actual) is 2.07 ETH, which is approximately 14 times lower than that of Scenario B (28.46 ETH). This case study demonstrates that strategic use of proxy contracts can substantially reduce deployment costs at scale.*

A.4 Largest Group of Type-1 Contracts per Clone Cluster

Given the number of contracts in the top 10 largest clusters, we decided to examine these clusters in greater detail. We refer to the largest clone cluster as Cluster 1, the second-largest as Cluster 2, and so on. To assess the homogeneity of each cluster, we determine the proportion of contracts that belong to the largest group of identical contracts (i.e., type-1 clones) within each cluster. We identify these groups by computing the MD5 hash of each contract and grouping those with identical hashes.

Our findings show that this proportion is 100% for Clusters 2, 3, and 8; 99.5% for Cluster 1; 99.2% for Cluster 6; 98.6% for Clusters 5 and 7; and 74.4%, 71.0%, and 29.3% for Clusters 9, 10, and 4, respectively (Table A.1). In other words, most of the top clusters predominantly consist of identical contracts (type-1 clones).

Table A.1: Distribution of type-1 clones across the top 10 clone clusters.

Top-10 Clusters	Number of Type-1 Clone Groups	Total Number of Contracts	Share of the Largest Type-1 Clone Group
Cluster 1	10	7138	99.5%
Cluster 2	1	502	100.0%
Cluster 3	1	405	100.0%
Cluster 4	6	242	29.3%
Cluster 5	3	147	98.6%
Cluster 6	2	119	99.2%
Cluster 7	2	71	98.6%
Cluster 8	1	54	100.0%
Cluster 9	6	43	74.4%
Cluster 10	2	31	71.0%

Given the high degree of homogeneity in the top 10 clusters, we select a *representative* contract from each cluster and briefly describe its purpose. This characterization offers insights into the types of SATD-related contracts that are frequently cloned. The representative contract of a cluster C is any contract that belongs to the largest group of identical contracts within C.

• **Cluster 1**: Authored by Gavin Wood (Ethereum co-founder), this contract implements a *multi-signature wallet*. In essence, a multi-signature wallet requires approvals from multiple parties before funds can be transferred. https://etherscan.io/address/0x6ae112b39805d5d4a1c807c2d8e8c554607ee110#code

• **Cluster 2**: Implements the *Proxy* design pattern (Gamma et al. 1994), allowing internal contract dependencies to be reconfigured at runtime (e.g., to support upgradeable contracts). https://etherscan.io/address/0x538fae85a5aad094148bee8b75643d2f0
code

- **Cluster 3**: Part of the AragonOS[1] framework, which offers reusable contracts for building decentralized organizations, DApps, and protocols. https://etherscan.io/address/0x3b20e426413ec06cedf1674e1bb671b6bf8cfd9b#code

- **Cluster 4**: Implements core functionality for a decentralized gambling application. https://etherscan.io/address/0xd9e8a52f712b0273770581f33b766a76724408d3#code

- **Cluster 5**: A component of a decentralized betting platform that lets users place bets on sports and esports events. https://etherscan.io/address/0x8021338f306fea03cf3152

code

- **Cluster 6**: Implements core functionality for a decentralized exchange (DEX) where users can buy and sell cryptocurrencies or tokens. https://etherscan.io/address/0xdf72b12a5f7f5a02e9949c475a8d90694d10f198#code

- **Cluster 7**: Similar to the multi-signature wallet in Cluster 1, with minor variations. https://etherscan.io/address/0x05276c19f670ffe7ba6b3bdc94e70c36b4b9dad5#code

- **Cluster 8**: A follow-up version of the contract described in Cluster 4. https://etherscan.io/address/0x1d896c9da6a32245cd035699dedaffd7f06f832c#code

- **Cluster 9**: Combines an upgradeable proxy mechanism with administrative access controls. https://etherscan.io/address/0x6de037ef9ad2725eb40118bb1702ebb27e4aeb2

code

[1] https://hack.aragon.org/docs/aragonos-intro.html

• **Cluster 10**: A standard ERC-20 token implementation. These tokens are commonly used as digital assets that users can hold and trade. https://etherscan.io/address/0x4204c9d1546c7a1394faad62f1ff249d355020e3#code

A.5 Examples of DApps for the Top-5 On-Chain Creational Patterns

Approach. The substantially large number of proxies (99.3%) that are systematically deployed using on-chain factories motivated us to examine some examples of large decentralized systems that employ such patterns. Thus, we attempted to find the largest DApps associated with the top-5 on-chain creational patterns (i.e., the second to sixth rows of Table 2). For each of the top-5 patterns, we reused a method similar to the use case viewpoint analysis in Section 5.5.2, and then picked the largest use case (a.k.a., the connected component) in terms of the number of proxy contracts. After that, we used Etherscan to manually identify the underlying system of the largest context. More specifically, we inspected the source/bytecode of the proxy contracts, logic contracts, and the contracts involved in the sequence through which the proxy is created. In particular, large-scale systems often utilize the name of the decentralized system as a pseudonym for the deployer address, which facilitates our identification in certain cases. Below, we discuss each pattern in the context of its corresponding system.

1. **EOA→FA→P**. We identified 6,618,012 proxy contracts for this pattern. The largest use case we found for this pattern is Opensea[2] with 943,022 proxies (14.2%).

[2] https://opensea.io

Opensea is the most prominent digital marketplace for crypto collectibles, especially non-fungible tokens (NFTs). Opensea uses a simple factory contract (WyvernProxyRegistry) to deploy one proxy per user, allowing them to have control over their assets. Each item that is traded on Opensea is owned by the proxy smart contract of a user. The corresponding sequence of contract names is as follows. Concerning the proxy (factory) contracts, the proxy contract name and its logic name are presented as a pair (e.g., [proxy contract name, logic contract name]).

EOA→FA [WyvernProxyRegistry[3]]→P [OwnableDelegateProxy[4], AuthenticatedProxy[5]]

2. **EOA→PF→P**. We identified 379,293 proxy contracts for this pattern. The largest use case we found for this pattern is Kraken[6] with 202,151 proxies (53.3%). Kraken is a decentralized exchange that provides a trading platform for buying, selling, and trading crypto assets. Kraken uses a proxy factory contract whose logic can be updated at runtime. Using this proxy factory, Kraken deploys a *minimalistic proxy* for each user Peter Murray 2018b. This minimalistic proxy merely forwards calls to a logic contract. Interestingly, the logic contract itself is a proxy contract that conducts several input validation checks before forwarding the calls to the final logic contract. Therefore, proxies here act as gates that validate the transaction input data. Our findings also show that there exist complex systems that use several layers of proxies stacked on top of each other. The corresponding sequence of contract names is as follows.

[3] https://etherscan.io/address/0xa5409ec958c83c3f309868babaca7c86dcb077c1
[4] https://etherscan.io/address/0xf69f020053685724feda8e617486442e08995607
[5] https://etherscan.io/address/0xf9e266af4bca5890e2781812cc6a6e89495a79f2
[6] https://www.kraken.com

EOA→PF [Non Verified[7], Not Verified[8]]→P [Not Verified[9], Not Verified[10]]

3. **EOA→FA→FA→P**. We identified 123,349 proxy contracts for this pattern. The largest use case we found for this pattern is the GnosisSafe Wallet DApp[11], which contains 51,746 (42%) proxy contracts. The corresponding sequence of contract names is as follows.

EOA→FA [Generic Factory[12]]→FA [GnosisSafeProxyFactory[13]]→P [GnosisSafeProxy[14], GnosisSafe[15]]

The instances of GnosisSafeProxy contracts are deployed by the GnosisSafeProxyFactory, which instantiates one instance per user request. We found that developers used another FA to deploy the GnosisSafeProxyFactory itself. We further analyzed why developers used another FA to deploy the GnosisSafeProxyFactory and found that the factory used for this purpose is a generic factory with minimalistic code. It only receives bytecode and then deploys it using the `create2` statement. Therefore, the chain becomes longer because developers preferred to use an on-chain generic factory to deploy the specialized GnosisSafeProxyFactory instead of off-chain scripts. There can be several reasons for using such a generic on-chain factory, including the determinism offered by the `create2` opcode, ensuring consistency across platforms and Ethereum forks, and simplifying deployment logic for contract reusability.

[7] https://etherscan.io/address/0xf9e266af4bca5890e2781812cc6a6e89495a79f2
[8] https://etherscan.io/address/0x2febe3d60bb2bbdab135c740617241c2eb949635
[9] https://etherscan.io/address/0x439039fbbbe5fd1de44fcacca694ac0da197a624
[10] https://etherscan.io/address/0x83b76b11257c4ece35370b6152f1946d49479e89
[11] https://safe.global/
[12] https://etherscan.io/address/0x4e59b44847b379578588920ca78fbf26c0b4956c
[13] https://etherscan.io/address/0xa6b71e26c5e0845f74c812102ca7114b6a896ab2
[14] https://etherscan.io/address/0xec55b27ffce0ed7c1de6568d98ad9a3b6219b6a7
[15] https://etherscan.io/address/0xd9db270c1b5e3bd161e8c8503c55ceabee709552

4. **EOA→FA→FA→FA→FA→P**. We identified 35,764 proxy contracts for this pattern, primarily associated with the Dharma Wallet DApp[16], with 18,397 proxy contracts. The corresponding sequence of contract names is as follows:

EOA→FA [Generic Factory[17]]→FA [ImmutableCreate2Factory[18]]→FA [ImmutableCreate2Factory[19]]→FA [DharmaSmartWalletFactoryV1[20]]→P [UpgradeBeaconProxyV1[21], DharmaSmartWalletImplementationV15[22]]

In this example, an EOA deployed a generic factory contract, followed by the deployment of the ImmutableCreate2Factory using the generic factory. ImmutableCreate2Factory offers secure contract deployment with a salt value and initialization code, preventing redeployment, collisions, and front-running. Another instance of ImmutableCreate2Factory is deployed through the former ImmutableCreate2Factory, and eventually, the DharmaSmartWalletFactoryV1 factory is deployed via the latter ImmutableCreate2Factory, which deploys clones of UpgradeBeaconProxyV1 proxy contracts. We further analyzed who initiated each of the four factory contracts' creation transactions:

- EOA initiated the Generic Factory (FA) creation transaction.
- EOA2 initiated the first ImmutableCreate2Factory (FA) creation transaction.
- EOA2 also initiated the second instance of the ImmutableCreate2Factory creation transaction.

[16] https://www.dharma.io/
[17] https://etherscan.io/address/0x7a0d94f55792c434d74a40883c6ed8545e406d12
[18] https://etherscan.io/address/0xcfa3a7637547094ff06246817a35b8333c315196
[19] https://etherscan.io/address/0x0000000000ffe8b47b3e2130213b802212439497
[20] https://etherscan.io/address/0xfc00c80b0000007f73004edb00094cad80626d8d
[21] https://etherscan.io/address/0x0004c95f9ba50a1ea11544565b71fab5dc5658c0
[22] https://etherscan.io/address/0x4d90ca10f218ebd4509ca0f9816cf20fd9903c35

- Dharma initiated the DharmaSmartWalletFactoryV1 creation transaction.

Given this analysis, it is evident that three different EOAs initiated the creation transactions of the intermediary factory contracts. Assuming that each EOA represents an independent developer, this observation highlights the practice of developers/practitioners reusing factory contracts deployed by others for their own purposes, which can lead to a higher pattern length.

5. **EOA→PF→PF→P**. We identified 1,968 proxy contracts, primarily associated with the PlotX DApp[23], with 720 (36.5%) proxy contracts. The corresponding sequence of contract names is as follows:

EOA→PF [OwnedUpgradeabilityProxy[24], Master[25]]→PF [OwnedUpgradeabilityProxy[26], MarketRegistryNew[27]]→P [OwnedUpgradeabilityProxy[28], Market[29]]

In this instance, the EOA deploys the first OwnedUpgradeabilityProxy "PF" contract, enabling upgradeability for the Master logic contract, which serves as a central registry and controller for other contracts within the PlotX DApp. The second OwnedUpgradeabilityProxy "PF" contract enables upgradeability for the MarketRegistryNew logic contract, primarily dealing with market creation and reward distribution. The MarketRegistryNew contract deploys Market instances by instantiating another OwnedUpgradeabilityProxy proxy contract that delegates to a Market logic contract. Our analysis revealed that all of these intermediary "PF" contracts were initiated by the PlotX organization, indicating the use of multiple layers of

[23] https://plotx.io/
[24] https://etherscan.io/address/0x03c41c5aff6d541ef7d4c51c8b2e32a5d4427275
[25] https://etherscan.io/address/0x4eb82f80858e6974307458b057b875bf3f23c4a0
[26] https://etherscan.io/address/0xe210330d6768030e816d223836335079c7a0c851
[27] https://etherscan.io/address/0x495d3a0530367ed4331833eae74b32d4848401f0
[28] https://etherscan.io/address/0x4c7861e96dc5a6e5acb31833ea5722e800a7bd24
[29] https://etherscan.io/address/0x25cf9d73b711bff4d3445a0f7f2e63ade5133e67

upgradeability proxy contracts for managing and upgrading their DApps' contracts at different levels.