Immutable Yet Mutable: Empirical Studies in Smart Contract Upgradeability

by

Ilham Ahmed Qasse

Thesis submitted to the Department of Computer Science
at Reykjavík University in fulfillment
of the requirements for the degree of
Doctor of Philosophy (Ph.D.)

November 2024

Thesis Committee:

Mohammad Hamdaqa, Supervisor
Assistant Professor, Reykjavík University, Polytechnique Montreal, Iceland, Canada

Björn Þór Jónsson, Co-Supervisor
Professor, Reykjavík University, Iceland

Kaiwen Zhang, Thesis Committee Member
Associate Professor, École de Technologie Supérieure ÉTS, Canada

Venera Arnaoudova, Thesis Committee Member
Associate Professor, Washington State University, United States

ISBN: 978-9935-539-55-7

ORCID: https://orcid.org/0000-0001-7526-3583

Immutable Yet Mutable: Empirical Studies in Smart Contract Upgradeability

Ilham Ahmed Qasse

November 2024

Abstract

Smart contracts have the potential to revolutionize business processes by digitizing agreements, reducing transaction costs, and enhancing security. However, while enabling trust and reliability, their immutable nature poses significant challenges when updates are necessary to address bugs or adapt to evolving requirements. Despite the introduction of various upgrade mechanisms, there remains a lack of comprehensive understanding of their characteristics, practical usage, limitations, and impact on security and user trust. Moreover, empirical studies analyzing these upgrade practices' real-world adoption and security implications are scarce.

This Ph.D. thesis addressed these gaps by systematically examining smart contract upgrade approaches, focusing on clarifying definitions, assessing practical usage, identifying limitations, and evaluating security implications. The core contributions of this thesis were as follows: (i) conducted a Multivocal Literature Review (MLR) to identify and systematically classify existing smart contract upgrade approaches, their characteristics, benefits, limitations, and best practices; (ii) developed PROXiFY, a bytecode-based tool that detects and classifies upgradeable contracts, assisting developers and users in identifying upgradeability features; (iii) performed a large-scale empirical analysis of over 44 million Ethereum contracts to evaluate the prevalence and practical adoption of upgrade approaches; (iv) analyzed the security implications of upgrades, examining how they addressed vulnerabilities and introduced new risks; and (v) introduced EvoChain, a framework for tracking smart contract evolution and upgrade history, enhancing transparency and supporting security assessments.

The findings of this thesis provide a comprehensive understanding of smart contract upgradeability. The MLR identified 17 unique upgrade approaches, which were categorized into structured classifications, each analyzed for their characteristics, benefits, and limitations. The empirical analysis demonstrated that only 3% of Ethereum contracts were upgradeable, yet these contracts collectively managed substantial economic value, underscoring their importance in the blockchain ecosystem. The security evaluation revealed that while 38% of upgrades effectively mitigated vulnerabilities, 62% either failed to address existing issues or introduced new vulnerabilities, highlighting the challenges of balancing adaptability with security. These findings provide actionable insights and practical tools to improve smart contract upgrade practices' transparency, security, and reliability.

Keywords: Smart Contract, Upgradeability, Blockchain Technology, Immutability, Ethereum, Software Maintenance, Proxy Patterns, Decentralized Applications (DApps)

Acknowledgments

I am deeply grateful to my supervisor, Dr. Mohammad Hamdaqa, whose guidance, support, and commitment have shaped my PhD journey. From our first meeting, Dr. Hamdaqa promised to provide the support I needed to grow and succeed, and he has honored that promise every step of the way. He has been there through all the ups and downs, offering not only academic advice but also steady encouragement as I adjusted to a new country and overcame challenges. Whether it was a critical PhD milestone or a draft needing another revision, he always made time to listen, guide, and inspire me to strive for excellence. His mentorship has helped me grow as both a researcher and an individual, and I am truly grateful for his support throughout this journey.

To my co-supervisor, Prof. Björn Þór Jónsson, thank you for your insights, mentorship, and guidance. I learned so much from him, especially in refining how I present and communicate ideas, focusing on clarity and precision, which have strengthened my work. His practical advice and feedback helped me navigate complex challenges, and – of course – his LaTeX tricks kept my documents running smoothly! Beyond technical advice, Prof. Jónsson's support and perspective have been invaluable, and I am grateful for the many ways he contributed to my PhD experience.

A special thank you to Dr. Manar Abu Talib, my master's co-supervisor, who played a crucial role in my decision to pursue a PhD. Her encouragement and belief in my potential gave me the confidence to take on this journey, and her support has remained steady and invaluable throughout.

To my thesis committee, thank you for your insightful comments and constructive feedback, which have added much depth to this work. I am grateful for the time and expertise each of you brought to my research.

To my co-authors, thank you for the inspiring collaborations and shared insights. Working with each of you has been a rewarding experience, and I am grateful for the knowledge and perspectives we built together.

To my friends, Naizeth and Vasiliki, thank you for being the best listeners and supporting me in countless ways, from emotional support and words of encouragement to cooking, feeding, and even motivating me to exercise during challenging times. To Isra, thank you for the countless idea-sharing sessions and long remote working sessions that started in the COVID era and continue beyond. To Shalini and Ioana, thank you for the joy you brought with board game nights, potlucks, and shared meals – these moments of fun and friendship have been priceless.

Finally, to my family, whose love and support have been my foundation: to my siblings, thank you for being my constant source of comfort and the perfect escape whenever I needed a change of scenery. And to my sister Suad, thank you for always being a phone call away; I truly couldn't have made it through without you.

Dedication

To my beloved parents, whose love and support have been my greatest blessings. To my father, my unwavering supporter and best friend, and to my mother, whose encouragement and faith in me have been my guiding light. Your strength and belief in my dreams have made this journey possible.

And to my second father, my dear uncle, whose absence is deeply felt. Though you are not here to witness this achievement, your encouragement to pursue my studies and your faith in my abilities continue to inspire me every day.

This thesis is dedicated to you all, with profound gratitude and love.

List of Publications

Appended publications

This thesis is based on the following publications:

1. **Paper A:** Ilham Qasse, Isra M. Ali, Nafisa Ahmed, Mohammad Hamdaqa, Björn Þór Jónsson, *The Myth of Immutability: A Multivocal Review on Smart Contract Upgradeability*, Submitted to Information and Software Technology (IST) Journal.

2. **Paper B:** Ilham Qasse, Mohammad Hamdaqa, Björn Þór Jónsson, *PROXiFY: A Bytecode Analysis Tool for Detecting and Classifying Proxy Contracts in Ethereum Smart Contracts*, Submitted to the Demonstrations Track of the 47th International Conference on Software Engineering (ICSE 2025).

3. **Paper C:** Ilham Qasse, Mohammad Hamdaqa, Björn Þór Jónsson, *Immutable in Principle, Upgradeable by Design: Exploratory Study of Smart Contract Upgradeability*, Second-round revisions submitted to Empirical Software Engineering (EMSE).

    - Ilham Qasse, Mohammad Hamdaqa, Björn Þór Jónsson, *Smart Contract Upgradeability on the Ethereum Blockchain Platform: An Exploratory Study*, In Proceedings of the 20th International Conference on Mining Software Repositories, Registered Reports (RR) track, 2023.

4. **Paper D:** Ilham Qasse, Mohammad Hamdaqa, Björn Þór Jónsson, *Beyond the Upgrade: An Empirical Study of Smart Contract Upgrades and Their Impact on Security*, Submitted to ACM Transactions on Software Engineering and Methodology (TOSEM) Journal (Under Review).

5. **Paper E:** Ilham Qasse, Mohammad Hamdaqa, Björn Þór Jónsson, *EvoChain: A Framework for Tracking and Visualizing Smart Contract Evolution*, Submitted to the Data and Tool Showcase Track of the 22nd International Conference on Mining Software Repositories (MSR 2025).

Other Publications

The following publications were completed during my PhD studies and are either published, or under review. However, they were not appended to this thesis, as their content does not directly relate to the thesis topic.

1. Mohammad Hamdaqa, Lucas Alberto Pineda Metz, and Ilham Qasse. 2020. *IContractML: A Domain-Specific Language for Modeling and Deploying Smart Contracts onto Multiple Blockchain Platforms.*, In Proceedings of the 12th System Analysis and Modelling Conference (SAM '20). Association for Computing Machinery, New York, NY, USA, 34–43. https://doi.org/10.1145/3419804.3421454

2. Mohammad Hamdaqa, Lucas Alberto Pineda Met, Ilham Qasse, *iContractML 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms*, Information and Software Technology, Volume 144, 2022, 106762, ISSN 0950-5849, https://doi.org/10.1016/j.infsof.2021.106762.

3. Ilham Qasse, Shailesh Mishra, Mohammad Hamdaqa, *iContractBot: A Chatbot for Smart Contracts' Specification and Code Generation,* 2021 IEEE/ACM Third International Workshop on Bots in Software Engineering (BotSE), Madrid, Spain, 2021, pp. 35-38, doi: 10.1109/BotSE52550.2021.00015.

4. Ilham Qasse, Shailesh Mishra, Björn Þór Jónsson, Foutse Khomh, and Mohammad Hamdaqa, *Chat2Code: A Chatbot for Model Specification and Code Generation, The Case of Smart Contracts,* 2023 IEEE International Conference on Software Services Engineering (SSE), Chicago, IL, USA, 2023, pp. 50-60, doi: 10.1109/SSE60056.2023.00018.

5. Majd Soud, Ilham Qasse, Grischa Liebel, Mohammad Hamdaqa, *AutoMESC: Automatic Framework for Mining and Classifying Ethereum Smart Contract Vulnerabilities and Their Fixes,* 2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Durres, Albania, 2023, pp. 410-417, doi: 10.1109/SEAA60479.2023.00068.

6. Mohamed Salah Bouafif, Chen Zheng, Ilham Qasse, Ed Zulkoski, Mohammad Hamdaqa, Foutse Khomh, *A Context-Driven Approach for Co-Auditing Smart Contracts with The Support of GPT-4 code interpreter*, Submitted to Information and Software Technology Journal.

Contents

Chapter 1

Introduction

"Change is the only constant in life." – **Heraclitus**

In smart contracts, the need for change presents a significant challenge. While immutability provides trust and reliability, it also poses obstacles when updates or modifications are necessary. This creates a critical dilemma: To upgrade or not? Can smart contracts be safely modified without compromising trust?

By analyzing over 44 million smart contracts on the Ethereum blockchain, this thesis uncovers key patterns, trends, and the impact of upgrades on security. Our work bridges the gap between theory and practice, offering insights and practical guidelines to help developers and researchers navigate this dilemma, paving the way for safer, more adaptable smart contracts.

1.1 Motivation

Blockchain technology has transformed digital transactions by establishing a decentralized, transparent, and secure ledger system [1–4]. Central to this technology are smart contracts, self-executing programs that enforce agreements automatically when predefined conditions are met [1–4]. Designed to be immutable, these contracts cannot be altered once deployed on the blockchain [1, 2, 5]. This immutability is essential for ensuring trust, security, and reliability within decentralized systems [1–3]. However, this very immutability also presents significant challenges, especially when vulnerabilities or bugs are identified after deployment [5–8]. Unlike traditional software, where updates and patches are routinely applied to resolve issues [9, 10], smart contracts remain unmodifiable, leaving them exposed to potential exploits [5–7]. This rigidity has led to high-profile incidents that reveal the risks and limitations of relying on immutable contracts [11].

1.1.1 Challenges and Necessity for Upgrading Smart Contracts

The Decentralized Autonomous Organization (DAO) attack in 2016 highlighted the risks posed by immutability [11–13]. The DAO, an investment fund operating via smart contracts on Ethereum, was exploited through a vulnerability that allowed an attacker to drain approximately $60 million worth of Ether. Since the contract was immutable, developers could not patch the vulnerability directly, which ultimately led

to a controversial hard fork to recover the stolen funds [14, 15]. This incident underscored the severe consequences of immutability when critical security issues emerge, raising questions about the flexibility of smart contracts in responding to threats [8, 14, 15].

In a similar incident in 2017, the Parity wallet freeze led to the permanent loss of over $150 million worth of Ether due to a flaw in a multi-signature wallet contract.[1] The inability to reverse the action due to the contract's immutability exemplified how unintended interactions with smart contracts can have irreversible financial consequences.

These high-profile incidents are not isolated cases. Several research studies have shown that many deployed smart contracts contain vulnerabilities [5–8]. For instance, He et al. [5] found that a significant number are vulnerable to common issues such as reentrancy attacks, integer overflows, and unauthorized access. This inability to update or patch these contracts post-deployment leaves them perpetually exposed, posing risks to users and the broader blockchain ecosystem [5–8].

These challenges have highlighted the need for mechanisms that allow smart contracts to adapt and evolve securely, ensuring that they can address critical vulnerabilities and improve functionality without compromising the core principles of trust and decentralization [16–18].

1.1.2 Existing Upgrade Approaches

To address the challenges of immutability, the blockchain community has developed several upgrade mechanisms that allow smart contracts to evolve over time without compromising their trustless[2] and secure nature [16–18]. These mechanisms enable solutions such as critical security patches, functional improvements, legal compliance, and performance optimizations [6, 16–18]. However, they also introduce complexities and potential vulnerabilities that must be carefully managed [8]. Key upgrade approaches include:

- **Contract Migration**: This approach involves deploying a new contract to replace an existing one. Users must manually interact with the new contract, which often requires migrating the state from the old contract [17, 18].

- **Proxy Patterns**: These patterns use a proxy contract to delegate calls to an implementation contract. The proxy holds the contract's state, while the logic resides in the implementation contract, which can be updated as needed [17, 19–21]. Examples include the Transparent Proxy Pattern and the Universal Upgradeable Proxy Standard (UUPS) [20, 21].

- **Diamond Standard (EIP-2535)**: This standard allows for modular contract designs, enabling updates to specific functions without affecting the entire contract [17, 22].

[1] `https://www.zdnet.com/article/parity-shakes-up-wallet-audits-but-funds-remain-frozen/`

[2] The term 'trustless' refers to systems where participants do not need to trust each other or a central authority, as the system's rules and outcomes are enforced automatically by technology, such as blockchain.

1.1.3 Gaps and Challenges in Current Approaches

Despite the development of these mechanisms, several significant challenges and gaps limit the secure and effective use of upgradeable contracts [16–18].

- **Lack of Standardization and Documentation (G1)**: A major issue is the lack of standardized definitions, classifications, and documentation for upgrade mechanisms. Developers and practitioners often use different terminologies and approaches, leading to inconsistencies that complicate communication within the community and impede the development of best practices. This lack of standardization creates ambiguity in how upgrade mechanisms should be implemented, making it difficult to evaluate their security and reliability. Without clear guidelines, developers may adopt inconsistent practices or overlook potential vulnerabilities, increasing the risk of errors in implementation.

- **User Challenges in Identifying Upgradeable Contracts (G2)**: From the user's perspective, determining whether a smart contract is upgradeable can be challenging due to the lack of visible indicators or standardized methods for identifying upgradeability. Users may not be aware of a contract's upgradeability status, which affects their trust and decision-making. The possibility of changes after deployment raises concerns about the introduction of malicious code or undesirable alterations in the contract's behavior. This lack of transparency necessitates improved methods for detecting and classifying upgradeable contracts to enhance user trust within the blockchain ecosystem.

- **Limited Empirical Evidence on Real-World Usage (G3)**: Despite the theoretical models available, there is limited empirical research on how these upgrade mechanisms are utilized in practice. Questions about real-world adoption remain unanswered, such as the prevalence of upgradeable contracts, the frequency of actual upgrades, and the factors influencing developers' choices of specific upgrade patterns. Without empirical data, it is difficult to assess the effectiveness and reliability of current approaches, thereby hindering the development of standardized, reliable upgrade practices.

- **Security Implications of Upgradeability (G4)**: One of the primary motivations for enabling upgrades is to enhance security by addressing discovered vulnerabilities. However, it remains unclear whether upgrades genuinely improve security or if they inadvertently introduce new risks. For instance, the Parity Wallet incident illustrates how even well-intentioned upgrades can inadvertently lead to severe security issues.[3] Empirical data on the security implications of upgrades is scarce, leaving open questions about whether upgrades effectively resolve vulnerabilities or lead to a decline in security. Without systematic analysis, the community lacks a clear understanding of these risks.

- **Limited Tracking of Smart Contract Versions (G5)**: Another significant gap is the lack of mechanisms for tracking existing smart contract versions. Without proper version control and documentation, it becomes challenging to monitor changes, verify the integrity of upgrades, and ensure that all stakeholders

[3] https://www.zdnet.com/article/parity-shakes-up-wallet-audits-but-funds-remain-frozen/

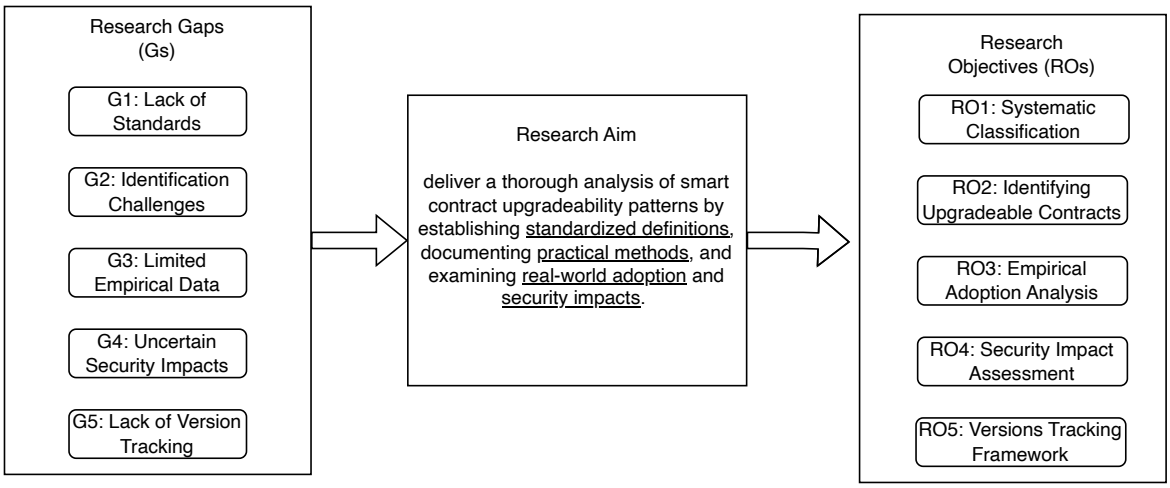

Figure 1.1: Mapping of identified research gaps (G1–G5) to the thesis aim and corresponding research objectives (RO1–RO5)

are aware of the current state of a contract. This absence of tracking hinders transparency and can lead to confusion or mistrust among users and developers.

To address these gaps, this Ph.D. thesis aims to *deliver a thorough analysis of smart contract upgradeability patterns by establishing standardized definitions, documenting practical methods, and examining real-world adoption and security impacts. Building upon this analysis, the study will develop a robust framework that provides developers and researchers with the necessary tools and guidelines to effectively manage the complexities of upgrading smart contracts while maintaining security and user trust.*

1.2 Research Objectives (ROs)

Building upon the identified research gaps, as discussed in Section 1.1.3, this thesis investigates smart contract upgradeability to enhance security, transparency, and reliability within blockchain applications, with a specific focus on Ethereum due to its open-source architecture, widespread adoption, and active community support [23–25]. The dissertation progresses from a broad examination of upgrade mechanisms to an in-depth analysis of proxy-based approaches, the most widely adopted upgrade pattern in Ethereum. To achieve the overall aim of this thesis, five research objectives (RO1–RO5) have been defined, each addressing a specific gap (G1–G5) and contributing to a systematic exploration of the problem space. These objectives are documented in dedicated chapters, which correspond to research papers that expand on the findings. Figure 1.1 illustrates the relationships between the research gaps, the thesis aim, and the objectives.

1.2.1 RO1: Systematic Classification and Analysis of Smart Contract Upgrade Approaches

The first objective is to systematically classify and analyze existing methods for upgrading smart contracts, with an emphasis on understanding their characteristics, benefits, and limitations. This objective addresses research gap G1, highlighting the lack of standardized definitions and classifications for upgrade mechanisms—a limitation

affecting consistency and best practices within the developer community. The results of this objective are discussed in Chapter 3 and correspond to Paper A.

To achieve this objective, the following research questions are addressed:

- **RQ1.1**: What are the existing approaches for upgrading Ethereum smart contracts?
- **RQ1.2**: What are the characteristics of each upgrading approach?
- **RQ1.3**: What are the benefits and limitations of each approach?

Methodology: To answer these questions, a Multivocal Literature Review (MLR) was conducted, synthesizing findings from both academic and grey literature to capture a comprehensive range of upgrade approaches. This approach enabled the integration of theoretical perspectives with practical insights from industry sources. Systematic searches were conducted in scholarly databases and Google (for grey literature) to identify upgrade approaches. Thematic analysis [26] was then used to categorize them based on technical attributes, including logic separation, state management, and upgrade processes. This analysis resulted in the development of a standardized taxonomy of smart contract upgradeability patterns, providing clarity and consistency for developers and researchers. The taxonomy revealed proxy-based mechanisms as the most widely adopted approach on Ethereum, guiding the focus of subsequent objectives toward this prevalent approach.

1.2.2 RO2: Developing a Tool for Detecting Proxy-Based Upgradeable Contracts

Addressing research gap G2, which centers on the challenges users face in identifying upgradeable contracts, this objective involves the development of a tool designed to detect and classify proxy-based upgradeable smart contracts. User difficulties in identifying upgradeable contracts affect transparency and trust in blockchain interactions, as developers and users often lack reliable means to determine a contract's upgradeability status. This objective is presented in Chapter 4 and corresponds to Paper B.

The research question guiding this objective is:

- **RQ2**: How can proxy-based upgradeable contracts be accurately detected and classified?

Methodology: Building on insights from RO1, we developed the PROXiFY tool utilizing bytecode analysis techniques to systematically detect and classify proxy-based upgradeable contracts. This process relies on identifying `delegatecall` instructions and tracing the associated implementation addresses, which form the basis for determining proxy-based upgradeability. The tool's functionality was evaluated on two datasets to evaluate its precision and recall, achieving high accuracy in detecting upgradeable proxies. By enabling developers and users to verify the upgradeability of contracts, the tool enhances transparency in Ethereum smart contracts, addressing the transparency challenges associated with proxy-based upgrades.

1.2.3 RO3: Empirical Analysis of Proxy-Based Upgrade Patterns in Ethereum

The third objective focuses on addressing research gap G3, which relates to the lack of empirical data on the real-world usage and adoption of upgrade mechanisms. By examining the prevalence and patterns of proxy-based upgradeable smart contracts on the Ethereum blockchain, this objective provides insights into how these approaches are applied in practice. The findings from this objective are presented in Chapter 5 and correspond to Paper C.

This objective is guided by the following research questions:

- **RQ3.1**: How prevalent are proxy-based upgrade patterns in Ethereum smart contracts?
- **RQ3.2**: How likely is an upgradeable contract to be upgraded?
- **RQ3.3**: What changes occur in smart contracts post-upgrade?
- **RQ3.4**: How does upgrading impact the activity level of smart contracts?

Methodology: Using the tool developed in RO2, approximately 44 million Ethereum contracts were analyzed to detect and classify proxy-based upgradeable contracts, allowing for an investigation into the adoption and usage patterns of proxy-based upgrades. Statistical analyses were conducted to assess adoption rates, upgrade frequencies, and usage scenarios, providing quantitative insights into upgrade behaviors. Additionally, a comparative analysis was performed to examine changes in contract functionality and user activity following upgrades. This empirical analysis offers valuable data-driven insights into the effectiveness of proxy-based upgrade mechanisms, informing best practices and strategies for secure and reliable upgrades in Ethereum applications.

1.2.4 RO4: Security Impact Analysis of Smart Contract Upgrades

The fourth objective addresses research gap G4, which focuses on the uncertain security implications of smart contract upgrades. Specifically, this objective examines whether upgrades enhance security or inadvertently introduce new vulnerabilities, providing a foundation for secure upgrade practices. The security analysis conducted for this objective is detailed in Chapter 6 and corresponds to Paper D.

This objective addresses the following research questions:

- **RQ4.1**: Is there a correlation between the number of versions a smart contract undergoes and overall security improvement or degradation?
- **RQ4.2**: How do smart contract upgrades impact contract security?
- **RQ4.3**: What are the most common vulnerabilities fixed while upgrading smart contracts?
- **RQ4.4**: What are the most common vulnerabilities introduced during upgrades?
- **RQ4.5**: Do some vulnerabilities persist across versions despite upgrade efforts?

Methodology: Leveraging the dataset collected in RO3, we conducted a detailed security analysis to explore the effects of upgrades on contract security. We analyzed the evolution of contract versions by comparing their Abstract Syntax Trees (ASTs), enabling tracking of changes in code structure across upgrades. Security analysis tools such as Slither [27] and Mythril[4] were employed to identify common vulnerabilities such as reentrancy and permission issues. Statistical methods were applied to identify trends in vulnerability persistence, introduction, and resolution, offering insights into the risks and benefits associated with smart contract upgrades. This objective provides insights into the security implications of upgradeable smart contracts, highlighting areas where security risks persist and the need for rigorous testing protocols during the upgrade process.

1.2.5 RO5: Developing a Framework for Tracking Smart Contract Evolution

The final objective addresses research gap G5, which highlights the lack of mechanisms for tracking smart contract versions, a gap that impedes transparency and complicates upgrade verification. We developed a framework to enable detailed tracking of smart contract evolution, capturing code changes and associated security assessments over time. The implementation of this framework is covered in Chapter 7 and corresponds to Paper E.

Methodology: Utilizing data from RO3 and RO4, we designed the EvoChain tool using Neo4J, a graph database that organizes the complex relationships and evolution of smart contracts. This tool models contract versions as interconnected nodes, capturing each version's code changes, upgrade events, and associated security issues. A user-friendly interface allows users to explore contract histories, upgrade paths, and security assessments, providing a granular view of contract evolution. By offering rich, accessible data, EvoChain supports transparency and trust in the blockchain ecosystem and empowers users to conduct further data-driven analyses to enhance secure upgrade practices.

1.2.6 Integration and Data Flow Between Objectives

The research objectives are strategically interconnected, creating a cohesive study where each objective builds upon the previous one to address the research gaps:

- RO1 establishes a theoretical foundation by categorizing upgrade mechanisms and identifying proxy-based patterns as predominant.
- RO2 leverages these insights to develop a tool for detecting proxy-based upgradeable contracts, essential for collecting empirical data in RO3.
- RO3 uses the tool to generate multiple datasets on real-world adoption and usage of upgradeable contracts.
- RO4 utilizes the versions dataset generated in RO3 for security analysis, investigating how upgrades impact security vulnerabilities over time.

[4] `https://github.com/ConsenSys/mythril`

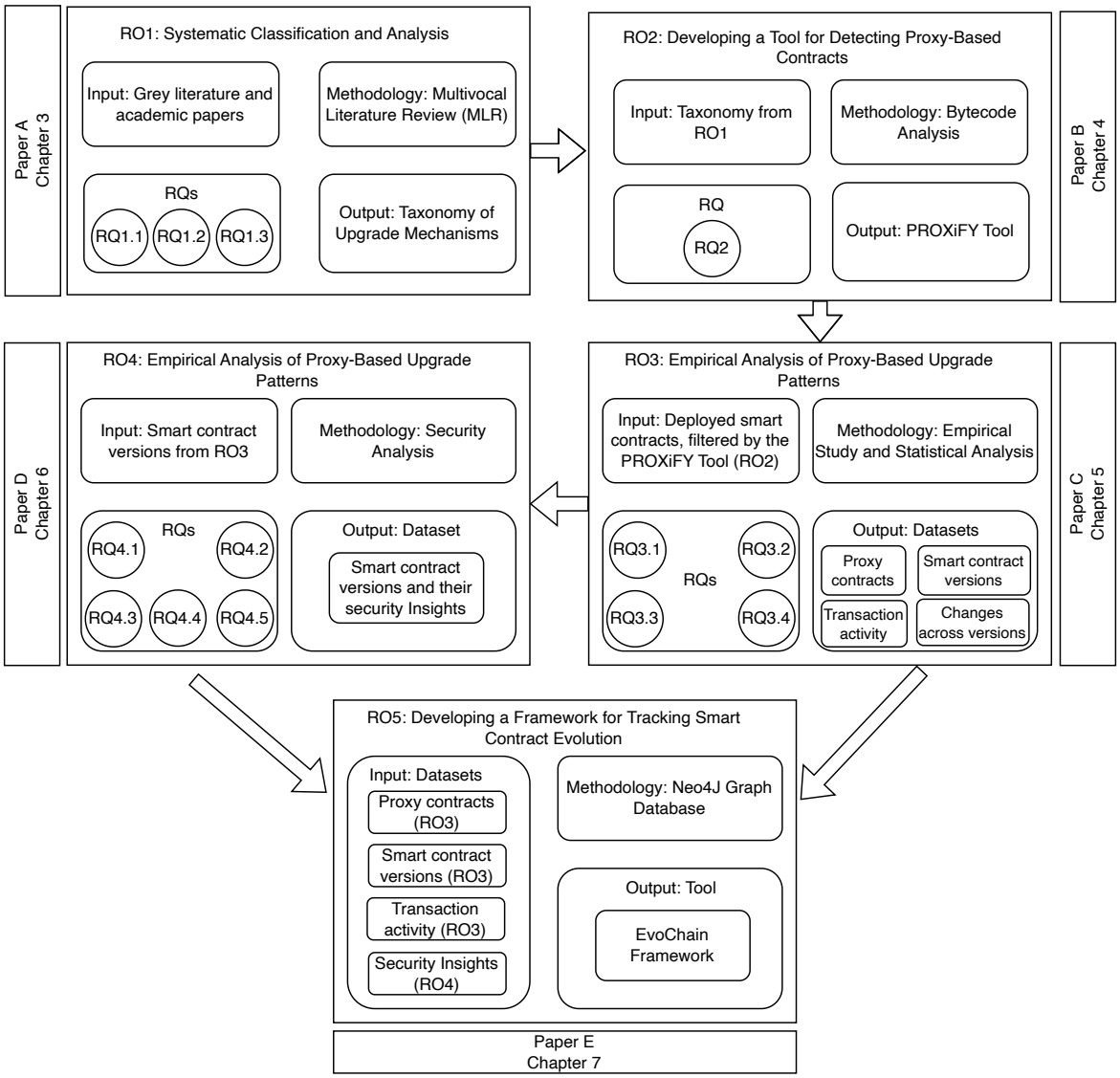

Figure 1.2: Overview of the data flow, methodologies, inputs, outputs, and interconnections among research objectives (RO1–RO5)

- RO5 integrates data from RO3 and RO4 to build a framework that tracks contract evolution, including code changes and security assessments.

Each research objective aligns with specific chapters and publications within the thesis, as shown in Figure 1.2, which illustrates the data flow, methodologies, inputs, and outputs across objectives. This figure highlights how each component contributes to a comprehensive analysis of smart contract upgradeability, methodology, and security considerations.

1.3 Thesis Contributions and Impact

This thesis significantly advances the field of smart contract upgradeability by addressing critical gaps in security, transparency, and management within blockchain applications, specifically focusing on Ethereum. The contributions are categorized into theoretical insights, empirical findings, and practical tools, each aligning with the

research objectives and corresponding studies. Collectively, they offer a comprehensive framework to guide the adoption and management of secure upgrade practices in Ethereum-based smart contracts..

- **Theoretical Contributions:**
  - Introduced a unified and comprehensive definition of smart contract upgradeability, bridging differing perspectives in the field and enhancing consistency across various upgrade approaches and contexts. This foundational definition provides clarity for developers and researchers, supporting a standardized understanding within the blockchain community (RO1).
  - Developed a structured taxonomy that categorizes upgrade approaches by technical attributes such as logic separation, state management, and upgrade processes. This structured taxonomy facilitates informed decision-making, allowing developers to effectively compare and select upgrade mechanisms that align with their specific requirements (RO1).
  - Conducted a comprehensive analysis of upgrade mechanism characteristics, benefits and limitations, focusing on attributes such as complexity, flexibility, security, and efficiency. This analysis provides practical guidance for developers seeking to implement upgradeable contracts that balance these attributes, offering actionable insights for effective design and implementation (RO1).
  - Identified best practices and common pitfalls in implementing smart contract upgrades, offering valuable recommendations for developers. Future research directions were also outlined, highlighting the importance of empirical studies in security, trust, and lifecycle management to support secure and transparent smart contract upgrades (RO1).
- **Empirical Contributions:**
  - Conducted a large-scale empirical study of approximately 44 million Ethereum contracts to examine the prevalence and application of proxy-based upgrade approaches. This extensive analysis reveals valuable insights into adoption rates, upgrade frequencies, and the real-world contexts in which upgrades are implemented, establishing a data-driven foundation for industry best practices (RO3).
  - Established a benchmark for Ethereum upgrade patterns by cataloging contract versions and tracking their evolution. This benchmarking provides developers, platform providers, and security auditors with a clear understanding of the practical adoption and effectiveness of various upgrade strategies, guiding more informed decision-making and best practices in Ethereum (RO3).
  - Introduced a novel dataset of smart contract vulnerabilities focused on their evolution over time, addressing a key gap in Ethereum security research. The dataset supports ongoing research by providing detailed insights into vulnerability persistence, resolution, and emergence across contract versions (RO4).

  - Conducted an empirical analysis of vulnerability evolution based on the vulnerability dataset to examine the impact of smart contract upgrades on security. By identifying patterns in vulnerability persistence and emergence, this analysis reveals trends that help developers anticipate potential security risks associated with upgrading, ultimately supporting secure upgrade practices (RO4).

- **Practical Tool Development:**

  - Developed PROXiFY, a tool designed to detect and classify proxy-based smart contracts without relying on source code or transaction history. PROXiFY addresses transparency challenges by enabling users and developers to verify a contract's upgradeability status, thereby improving trust and supporting informed decision-making within the Ethereum ecosystem (RO2).
  - Introduced EvoChain, a tool that allows developers, auditors, and other stakeholders to visualize contract histories, upgrade paths, and security assessments over time. This accessibility encourages wider adoption of secure upgrade practices and supports transparency by enabling stakeholders to verify upgrades and evaluate security status throughout a contract's lifecycle (RO5).

1.4 Organization of Thesis

The remainder of this thesis is organised as follows. Chapter 2 provides the background necessary for understanding the thesis's core concepts, including blockchain technology, Ethereum, smart contracts, and software evolution. Chapter 3 covers the first research objective (RO1) by providing an overview of existing smart contract upgrade approaches and discussing their benefits and limitations. This chapter also serves as the related work section for the thesis, providing essential context and background to support the subsequent research objectives. Chapter 4 addresses the second objective (RO2), describing the development of a tool to detect proxy-based upgradeable contracts aimed at improving transparency and trust in blockchain applications. Chapter 5 covers the third objective (RO3) by presenting an analysis of upgrade patterns in Ethereum, offering insights into how proxy-based approaches are used in practice. Chapter 6 addresses the fourth objective (RO4) by examining the security implications of smart contract upgrades, focusing on vulnerabilities that upgrades may introduce or fix. Chapter 7 covers the fifth objective (RO5) by presenting a framework for tracking smart contract evolution, allowing detailed tracking of code changes and security assessments over time. Finally, Chapter 8 concludes the thesis by summarizing the main contributions and impact of the research, discussing the limitations, and outlining directions for future research.

Chapter 2

Background

This chapter introduces foundational concepts related to this thesis's research objectives. Section 2.1 introduces blockchain technology, highlighting its core features such as decentralization, immutability, and consensus mechanisms. Section 2.2 focuses on the Ethereum blockchain, explaining its unique capabilities, including the Ethereum Virtual Machine (EVM), and its significance for this research. The chapter also discusses smart contracts in Section 2.3, covering their features, programming languages such as Solidity, and why Solidity is central to this study. Section 2.4 addresses the security challenges of smart contracts and methods for analyzing vulnerabilities. Finally, Section 2.5 explores software evolution and maintenance, emphasizing their relevance to blockchain systems and smart contracts.

2.1 Blockchain Technology

Blockchain technology is a decentralized ledger system that enables secure, transparent, and tamper-proof record-keeping of transactions across a distributed network [28]. Its core features include decentralization, immutability, transparency, security, consensus mechanisms, and programmability through smart contracts.

2.1.1 Decentralization

Decentralization eliminates the need for a central authority by distributing the ledger across multiple nodes in the network [29]. Each node maintains a copy of the blockchain, ensuring data redundancy and fault tolerance. This structure mitigates risks associated with centralized systems, such as single points of failure and centralized control [30].

2.1.2 Immutability

Immutability is achieved through cryptographic hashing and the chaining of blocks, where each block contains a hash of the previous block [31]. Altering any transaction would require changing all subsequent blocks and gaining control over the majority of the network's computational power, making tampering practically infeasible [32].

2.1.3 Transparency

Transparency allows all network participants to view transactions on the ledger, promoting accountability and auditability [33]. While transactions are visible, user identities can remain pseudonymous, balancing transparency with privacy [28].

2.1.4 Security

Blockchain security is enforced through cryptographic techniques, including digital signatures and hash functions [34]. Transactions are signed by the sender's private key, ensuring authenticity and non-repudiation [35].

2.1.5 Consensus Mechanisms

Consensus mechanisms enable network participants to agree on the state of the ledger. Common mechanisms include:

- **Proof of Work (PoW)**: Nodes solve complex mathematical puzzles to validate transactions (used by Bitcoin) [28].
- **Proof of Stake (PoS)**: Validation power is proportional to the number of tokens held (used by Ethereum 2.0) [36].

2.1.6 Blockchain Platforms

Blockchain platforms can be categorized into public and private, as well as permissioned and permissionless blockchain [37]. The categorization is based on the ability of a node to join or append a new block to the network [37]. In the public blockchain, any node can access transactions [38]. On the other hand, access to the network and transactions are restricted in the private blockchain [38]. In permissioned blockchain, participants need prior approval to join the network, while in permissionless blockchain anyone can participate [39, 40]. The network is maintained by a group of nodes in a permissioned blockchain.

2.2 Ethereum Blockchain Platform

Ethereum is a decentralized, open-source blockchain platform that introduced significant advancements in smart contract development [29]. Unlike Bitcoin, which primarily focuses on cryptocurrency transactions, Ethereum provides a programmable blockchain through the Ethereum Virtual Machine (EVM), enabling developers to build and deploy decentralized applications (DApps) [41].

2.2.1 Ethereum Virtual Machine (EVM)

The Ethereum Virtual Machine (EVM) is a Turing-complete virtual machine that executes smart contract bytecode on the Ethereum network [31]. It provides a sandboxed environment where smart contracts can run deterministically, ensuring that contract execution yields the same result on all nodes [42]. Smart contracts are written in high-level languages such as Solidity and compiled into bytecode that the EVM can interpret [43].

2.2.2 Gas Mechanism

Gas is a fundamental concept in Ethereum that measures the computational effort required to execute operations or transactions [41]. Each operation in the EVM has an associated gas cost, and users must specify a gas limit and gas price when initiating transactions [31]. If the gas runs out during execution, the transaction fails, but the miner keeps the gas fees for the computational work performed [44].

2.2.3 Rationale for Focusing on Ethereum

Ethereum is the focus of this thesis due to its unique and robust features that make it a leading platform for smart contract development. These include:

- **Active Developer Community**: Ethereum has one of the largest and most active developer communities in the blockchain space [45]. This community has contributed to a rich ecosystem of tools, libraries, and frameworks that facilitate efficient smart contract development and deployment [46].
- **Smart Contract Support**: Ethereum was designed with smart contracts as a core feature, enabling complex programmable transactions [41]. The EVM provides a robust execution environment for smart contracts, supporting a wide range of decentralized applications (DApps) [31].
- **Upgradeable Contracts**: Ethereum supports upgradeability patterns, allowing developers to modify contract logic after deployment [47]. This capability is essential for adapting to new requirements and fixing vulnerabilities, as discussed in Chapter 3.
- **Availability of Deployed Contracts**: Ethereum's vast repository of deployed smart contracts, all publicly accessible, makes it an ideal platform for empirical studies and data-driven analysis [48].

2.3 Smart Contracts

Smart contracts are programmable agreements that automatically execute when predefined conditions are met [49]. They facilitate, verify, and enforce the negotiation or performance of a contract without third parties [50].

2.3.1 Features of Smart Contracts

Smart contracts exhibit features such as:

- **Automation**: Execution occurs automatically upon meeting conditions, reducing manual intervention [51].
- **Trustlessness**: Parties do not need to trust each other; trust is placed in the code's correctness [29].
- **Transparency**: Contract code and execution are visible on the blockchain, enhancing accountability [31].

Listing 2.1: Solidity Sample Code

```
pragma solidity >=0.4.16 <0.9.0;
contract SimpleStorage {
    address public owner;
    uint256 storedData;
    constructor() {
        owner = msg.sender;
    }
    function set(uint256 x) public {
        require(msg.sender == owner);
        storedData = x;
    }
    function get() public view returns (uint) {
        return storedData;
    }
}
```

- **Immutability**: Once deployed, contracts cannot be altered, preventing unauthorized changes [44].

- **Efficiency**: The elimination of intermediaries reduces costs and speeds up transactions [49].

2.3.2 Solidity Programming Language

The Solidity language is an object-oriented language deployed on the EVM. In addition to Ethereum, several blockchain platforms support Solidity, including Quorum, Hyperledger Burrow, and Hyperledger Besu. Solidity has syntax similar to C and JavaScript, but it includes several unique concepts specific to smart contracts, including visibility of function modifiers: internal, external, view; emitted events; and smart contract-specific operations such as self-destruct and revert. Solidity supports various variable types, including int, bool, and string, as well as unique types such as mapping and address. A variable type mapping provides access to storage by mapping keys to values, similar to a dictionary in Python. Variable type address is designed to store Ethereum accounts' addresses and provide member functions such as balance, send, transfer, and call. A default value is always assigned to newly declared variables, depending on their type. For instance, a boolean variable is always initialized with false, while an integer is always initialized with zero. A sample of Solidity code is shown in Listing 2.1. The code shows a simple storage contract that allows only the owner to store an integer value.

The thesis focuses on Solidity due to the following reasons:

- **Primary Language for Ethereum**: Solidity is the most widely used language for writing Ethereum smart contracts [43].

- **Community Support**: A large developer community contributes to Solidity's libraries, tools, and best practices [45].

- **Comprehensive Tooling**: The Solidity ecosystem includes compilers, debuggers, and security analysis tools, which streamline development and ensure contract robustness [52].

- **Rich Feature Set**: Solidity supports advanced programming constructs such as inheritance, libraries, and modifiers, enabling the creation of complex and efficient smart contracts [31].

2.4 Smart Contract Analysis and Security

The security of smart contracts is paramount due to the high stakes involved in managing assets and executing critical operations on blockchain networks [53]. Analyzing smart contracts for vulnerabilities and ensuring their robustness requires a combination of methods, including abstract syntax tree (AST) analysis, bytecode analysis, and various security-focused approaches.

2.4.1 Analysis Methods

Smart contracts are analyzed using both source code and bytecode techniques to detect vulnerabilities and ensure compliance with security standards. AST analysis represents the syntactic structure of source code in a tree-like format, enabling static analysis tools to identify patterns associated with vulnerabilities such as reentrancy [52]. For example, tools such as SmartCheck parse Solidity code into an AST to systematically evaluate compliance with best practices [52]. Beyond security, AST analysis supports refactoring, optimization, and enforcing coding standards [54].

Bytecode analysis becomes crucial when source code is unavailable. This method examines the compiled low-level code executed by the Ethereum Virtual Machine (EVM) [55]. Techniques such as symbolic execution simulate contract behavior under various input conditions to uncover logical errors and vulnerabilities [56]. Disassembly, converting bytecode into an intermediate representation, allows for manual inspection of contract behavior [55]. However, challenges in bytecode analysis include the loss of high-level constructs during compilation, obfuscation to protect intellectual property, and optimizations by compilers that alter code flow [57]. Despite these challenges, bytecode analysis provides critical insights, especially for unverified contracts on public blockchains.

2.4.2 Security Considerations

Smart contract security is critical to prevent financial and operational risks. Vulnerabilities such as reentrancy attacks, where external recursive calls exploit delayed state updates, have resulted in significant financial losses, including the infamous DAO hack [11, 44]. Improper access control, allowing unauthorized execution of privileged functions, and integer overflows or underflows, causing incorrect calculations, are also common [53, 58]. Denial of Service (DoS) attacks, which exploit gas limits to disrupt contract functionality, further highlight the need for robust security measures [44].

Security analysis methods include static analysis, dynamic analysis, and formal verification. Static analysis tools such as Mythril and Slither evaluate contract code

without execution to identify vulnerabilities such as unprotected state variables or unsafe external calls [27]. Dynamic analysis, on the other hand, observes runtime behaviors through controlled execution environments, with tools such as Echidna performing fuzz testing to detect unexpected behavior [59]. Formal verification uses mathematical methods to rigorously prove that a contract behaves as intended, ensuring compliance with specified safety properties [60].

Mitigating these vulnerabilities involves secure coding practices, such as enforcing checks-effects-interactions patterns to prevent reentrancy [61]. Regular code audits by third-party experts are essential to identify and resolve hidden vulnerabilities [62]. Leveraging well-tested libraries, such as those provided by OpenZeppelin, reduces the likelihood of introducing errors in commonly used patterns [63]. Extensive testing and simulation using testnets further ensure the robustness of smart contracts before deployment [45].

2.5 Software Evolution and Maintenance

Software evolution refers to the process by which software systems are continuously modified and enhanced to adapt to new requirements, correct defects, improve performance, and maintain relevance in changing environments [64, 65]. It encompasses planned and unplanned changes that occur throughout the software lifecycle. Effective evolution ensures that software meets user needs, accommodates regulatory changes, and maintains operational efficiency [66].

In contrast, software maintenance focuses on modifying software programs after their release [67]. Maintenance activities typically fall into four categories:

- **Corrective Maintenance**: Fixing identified defects or errors.
- **Adaptive Maintenance**: Adjusting software to work in new or changing environments.
- **Perfective Maintenance**: Enhancing performance, usability, or other attributes.
- **Preventive Maintenance**: Addressing potential issues before they occur [68].

While maintenance implies preservation and addressing known issues, evolution considers broader scientific goals, including the creation of new patterns and behaviors arising from existing systems [69]. The differences are summarized as follows:

- Maintenance emphasizes fixing and preservation, while evolution involves broader, dynamic changes.
- Maintenance is typically planned, while evolution incorporates both planned and emergent changes.
- Maintenance focuses on engineering goals, whereas evolution aligns with scientific and long-term objectives [69].

2.5.1 Challenges in Smart Contract Evolution

Blockchain-based systems, particularly smart contracts, present unique challenges for evolution due to their immutable nature. Once deployed, smart contracts cannot be

directly modified, posing significant hurdles for updating functionality or correcting defects [44]. Alternative approaches, such as deploying new versions or employing upgradeability patterns such as proxies, increase complexity and dependency management issues [47, 70]. Additionally, decentralized environments lack centralized version control systems, complicating the tracking and maintenance of multiple versions.

These challenges necessitate robust planning and secure upgrade mechanisms to ensure smart contracts evolve effectively without compromising their integrity or security [61]. In this context, upgradeability patterns play a critical role, enabling the safe adaptation of smart contracts to new requirements. This topic is further explored in Chapter 3.

2.6 Summary

This chapter outlined the foundational concepts of blockchain technology, emphasizing decentralization, immutability, and transparency, focusing on Ethereum as the primary platform due to its programmability and support for upgradeability patterns. It introduced smart contracts, highlighting their features, the Solidity programming language, and challenges in security and evolution, including methods like static analysis and formal verification. Finally, the chapter addressed software evolution and maintenance, identifying upgradeability patterns essential for adapting blockchain systems while preserving security and transparency.

In the next chapter, we investigate the first research objective (RO1), providing a comprehensive overview of existing approaches to smart contract upgradeability and establishing a structured taxonomy for analyzing these mechanisms.

Chapter 3

The Myth of Immutability: A Multivocal Review on Smart Contract Upgradeability

Authors: Ilham Qasse, Isra M. Ali, Nafisa Ahmed, Mohammad Hamdaqa, Björn Þór Jónsson
Status: Submitted to Information and Software Technology (IST) Journal.

Highlights

- Defines smart contract upgradeability, uniting varied perspectives in the field.
- Presents a unified taxonomy and classification of smart contract upgrade methods.
- Examines the impact of upgrade approaches on core smart contract components.
- Analyzes the benefits and limitations of upgrade methods based on software quality attributes.
- Provides practical insights, best practices, and future research directions.

The Myth of Immutability: A Multivocal Review on Smart Contract Upgradeability

Ilham Qasse[a], Isra M. Ali[b,1], Nafisa Ahmed[c,1], Mohammad Hamdaqa[a,c], Björn Þór Jónsson[a]

[a]*Department of Computer Science, Reykjavik University, Reykjavik, Iceland*
[b]*College of Science and Engineering, Hamad Bin Khalifa University, Doha, Qatar*
[c]*Department of Computer Engineering and Software Engineering, Polytechnique Montreal, Montreal, Canada*

Abstract

Background: The immutability of smart contracts on blockchain platforms like Ethereum promotes security and trustworthiness but presents challenges for updates, bug fixes, or adding new features post-deployment. These limitations can lead to vulnerabilities and outdated functionality, impeding the evolution and maintenance of decentralized applications. Despite various upgrade mechanisms proposed in academic research and industry, a comprehensive analysis of their trade-offs and practical implications is lacking.
Aims: This study aims to systematically identify, classify, and evaluate existing smart contract upgrade mechanisms, bridging the gap between theoretical concepts and practical implementations. It introduces standardized terminology and evaluates the trade-offs of different approaches using software quality attributes.
Methods: We conducted a Multivocal Literature Review (MLR) to analyze upgrade mechanisms from both academic research and industry practice. We first establish a unified definition of smart contract upgradeability and identify core components essential for understanding the upgrade process. Based on this definition, we classify existing methods into full upgrade and partial

Email addresses: ilham20@ru.is (Ilham Qasse), https://orcid.org/0000-0002-2689-348X (Isra M. Ali), nafisa.abdelmutalab-ali-ahmed@polymtl.ca (Nafisa Ahmed), mhamdaqa@polymtl.ca (Mohammad Hamdaqa), mhamdaqa@ru.is (Mohammad Hamdaqa), bjorn@ru.is (Björn Þór Jónsson)
[1]These authors contributed equally to this work.

Preprint submitted to Information and Software Technology *January 25, 2025*

upgrade approaches, introducing standardized terminology to harmonize the diverse terms used in the literature. We then characterize each approach and assess its benefits and limitations using software quality attributes such as complexity, flexibility, security, and usability.
Results: The analysis highlights significant trade-offs among upgrade mechanisms, providing valuable insights into the benefits and limitations of each approach. These findings guide developers and researchers in selecting mechanisms tailored to specific project requirements.
Conclusions: By offering a comprehensive evaluation that bridges theoretical concepts and practical implementations, this study contributes to the development of more resilient, adaptable, and secure decentralized systems.

Keywords: Smart Contract, Upgradeability, Blockchain Technology, Immutability, Multivocal Literature Review, Ethereum, Software Maintenance, Proxy Patterns, Decentralized Applications (DApps), Upgrade Patterns, Software Quality Attributes

---

1. Introduction

Software systems constantly evolve to fix bugs, add new features, and adapt to changing user needs and technological environments [1, 2]. This ongoing evolution is fundamental to software engineering, ensuring that applications stay relevant, secure, and efficient over time [3]. Traditional software upgrade methods allow developers to update systems smoothly, minimizing user disruptions and maintaining system stability [4]. However, the advent of blockchain technology and smart contracts has introduced a new paradigm in software evolution, fundamentally challenging traditional notions of upgradeability.

Smart contracts are self-executing programs stored on a blockchain, designed to uphold the principles of immutability to ensure transparency, security, and trustworthiness [5]. Once deployed, a smart contract's code is permanent and cannot be altered, posing a significant challenge: *How can developers upgrade smart contracts without undermining the immutability that ensures their reliability* [6]? This inherent tension between the need for software evolution and the immutable nature of smart contracts has become a critical issue in blockchain development.

High-profile incidents, such as the infamous DAO attack [7, 8], where a vulnerability in a smart contract was exploited to drain over $60 million

worth of Ether, highlight the urgency of this problem. Vulnerabilities in deployed contracts expose the limitations of immutability, where the inability to make post-deployment modifications prevents fixing errors, adding new features, and meeting new regulations [9, 10]. These challenges are especially significant in Ethereum, the most widely used platform for smart contracts, since it supports a large ecosystem of decentralized applications (DApps) and decentralized finance (DeFi) protocols.

Several studies have explored aspects of smart contract upgradeability [11, 12, 13, 14, 15]. Salehi et al. [12] summarized and evaluated six upgradeability patterns, developing a framework to measure the prevalence of these patterns on the Ethereum blockchain. Bui et al. [15] analyzed the need for upgradability in smart contracts, thoroughly reviewed existing upgradeable patterns, identified their main differences and limitations, and introduced a new pattern to address them. However, these studies are often limited in scope, focusing on specific aspects such as the evaluation of certain patterns or the introduction of new solutions. There is still a lack of comprehensive analysis that integrates academic research and industry practices, resulting in inconsistent definitions and classifications and restricting the development of standardized, broadly applicable approaches.

To bridge this gap, we conduct a *Multivocal Literature Review* (MLR) that consolidates findings from both peer-reviewed academic sources and grey literature, such as technical blogs, reports, and community forums. The MLR approach is uniquely suited for integrating theoretical and practical perspectives, providing a holistic view that can inform standardized upgrade practices [16]. The main Research Objectives (ROs) of this study are to:

1. **RO1:** Classify existing approaches for upgrading Ethereum smart contracts, presenting a clear taxonomy that includes diverse methods from theory and practice.

2. **RO2:** Analyze the characteristics of each upgrading approach, focusing on essential smart contract components.

3. **RO3:** Assess the benefits and limitations of each approach in relation to standard software quality attributes.

To achieve these objectives, we systematically searched major academic databases and grey literature platforms to capture a broad spectrum of perspectives. Employing thematic analysis and framework analysis, we extracted

and synthesized data to comprehensively understand smart contract upgradeability [17, 18].

This paper contributes to software engineering and blockchain technology by (i) introducing a comprehensive definition of smart contract upgradeability, addressing differing perspectives in the field; (ii) establishing a standardized taxonomy of upgrade mechanisms, unifying terminology to enhance clarity and communication; (iii) analyzing and classifying the characteristics of upgrade mechanisms, providing insights into their impact on core smart contract components; (iv) evaluating the benefits and limitations of each approach based on software quality attributes, offering practical guidance for developers; and (v) highlighting best practices and common pitfalls, and outlining future research directions, emphasizing the need for empirical studies on security, trust, and lifecycle management of upgrade mechanisms.

The remainder of this paper is organized as follows: Section 2 provides background information on smart contracts and their key characteristics, with a focus on Ethereum. Section 3 explains the research methodology, including the search strategy, source selection, and data extraction. In Section 4, we systematize the main ideas related to upgradeability and present our unified definition. Section 5 covers RQ1 by classifying current upgrade methods. Section 6 addresses RQ2 by analyzing the features of each method. Section 7 covers RQ3 by evaluating the pros and cons of each method based on software quality factors. Section 8 discusses the findings, including governance models and lifecycle management. Section 9 explains threats to validity. Finally, Section 10 concludes the paper and suggests future research directions.

2. Background

Smart contracts are self-executing contracts, with the terms of the agreement directly written into code which operates on blockchain technology. They automatically enforce and execute contractual obligations when predefined conditions are met, eliminating the need for intermediaries [19]. This automation enhances efficiency and reduces the potential for disputes, making smart contracts a significant innovation in decentralized systems. Their importance lies in their ability to provide trusted transactions in trustless[2]

[2]The term 'trustless' refers to systems where participants do not need to trust each other or a central authority, as the system's rules and outcomes are enforced automatically

settings, streamline processes, and reduce administrative costs across various sectors, including finance, supply chain management, and healthcare [20, 3].

2.1. Core Characteristics and Applications

Smart contracts possess several key characteristics that are essential for their functionality:

- **Transparency**: All smart contract transactions are recorded on a blockchain, allowing all parties to verify the contract's execution and state. This transparency fosters trust among users [21, 16].
- **Immutability**: Smart contracts cannot be altered once deployed, ensuring that the agreed-upon terms remain intact. This characteristic prevents tampering and enhances security, although it also poses challenges for updates [22].
- **Autonomy**: Smart contracts operate independently without human intervention once they are deployed. This autonomy reduces the risk of human error and increases efficiency [23].
- **Security**: Smart contracts leverage cryptographic techniques to secure transactions and data. The decentralized nature of blockchain technology further enhances security by eliminating single points of failure [24].

These characteristics contribute to the reliability and effectiveness of smart contracts in various applications. For example, in Decentralized Finance (DeFi), smart contracts facilitate automated trading, lending, and borrowing without intermediaries, enhancing efficiency and reducing costs [3, 25]. In Supply Chain Management, they enable transparent tracking of goods and services, minimizing disputes and improving accountability [23, 21]. Additionally, voting systems utilize smart contracts to create secure and transparent processes, ensuring accurate vote counts and preventing tampering [21, 26].

---

by technology, such as blockchain.

2.2. Technical Considerations and Gas Costs

Gas is a unit of measurement for the computational effort required to execute operations within a blockchain network, such as Ethereum. Each operation performed by a smart contract consumes a specific amount of gas, which users pay for in the network's native cryptocurrency (e.g., Ether on Ethereum). Gas costs play a critical role in influencing the efficiency and usability of smart contracts, as high fees may deter users from engaging with blockchain applications [27]. Developers adopt strategies such as efficient code design and optimized data structures to minimize gas costs [27, 25].

Another technical consideration is the Self-Destruct Mechanism (`SELF DESTRUCT` operation). This operation allows a smart contract to remove its code and state from the blockchain, freeing up storage space. While useful for managing outdated contracts, it risks permanently losing state data, which can impact users relying on that information [22].

Execution Flow and Limitations involve managing function calls that consume gas and must stay within network-imposed gas limits. Exceeding these limits results in "Out of Gas" errors, causing transaction failures [3, 22]. Proper planning and code optimization are essential to prevent such issues.

2.3. Security Considerations

Security is paramount for smart contract functionality due to potential vulnerabilities, which could result in significant financial losses. Error Handling and Fallback Functions are crucial for maintaining contract stability. Fallback functions are triggered when a contract receives Ether without data or when an invalid function is called, helping manage unexpected conditions [3]. Mechanisms like `require`, `assert`, and `revert` ensure that contracts behave as intended and can recover gracefully from errors [22].

Reentrancy Attacks pose a significant security risk, as seen in the DAO incident, where an attacker exploited reentrancy to repeatedly call back into the contract before its execution was complete, resulting in substantial financial losses [28]. Implementing best practices such as the "checks-effects-interactions" pattern is essential for minimizing such risks [28]. This pattern improves security by structuring contract functions first to validate inputs ('checks'), then update the contract state ('effects'), and only afterwards perform external calls ('interactions'). By ensuring state changes occur before any external interactions, this approach prevents malicious contracts from exploiting reentrant calls to manipulate the contract's state.

2.4. Adaptability and Immutability

The balance between adaptability and immutability is a major challenge for smart contract developers. Immutability ensures trust and security by preventing changes after deployment, but it also restricts the ability to update contracts or fix errors. This limitation can impede the long-term adaptability of smart contracts in dynamic environments where changes are often necessary [22, 6]. Innovative upgrade mechanisms are crucial for maintaining functionality while preserving the core principles of blockchain, such as Transparency and user trust.

2.5. Focus on Ethereum

This study focuses on Ethereum due to its widespread adoption as the leading platform for smart contract development. Ethereum has a large, active community that provides extensive support for developers and researchers, contributing valuable resources and discussions [29, 30, 31]. This strong community presence and the platform's open-source nature make Ethereum an ideal choice for examining upgrade mechanisms through a comprehensive Multivocal Literature Review (MLR). The mature ecosystem of Ethereum and the availability of diverse academic and grey literature further support its selection for this study.

3. Methodology

This section presents the methodology for this MLR, following the guidelines outlined by Garousi et al. [32] for conducting Multivocal Literature Reviews (MLRs) and integrating grey literature in software engineering. This structured approach ensures a comprehensive analysis of smart contract upgrade mechanisms by combining academic literature and grey literature (GL). Figure 1 summarizes the methodology, highlighting key steps such as source selection, quality assessment, and data extraction, which are described in detail in the remainder of this section.

3.1. Planning the Multivocal Literature Review (MLR)

The first step in the MLR process, as outlined by Garousi et al. [32], is the planning phase. This involves defining the need for an MLR and formulating clear research questions to guide the review. This section details the rationale for this approach and the research questions that shape the study.

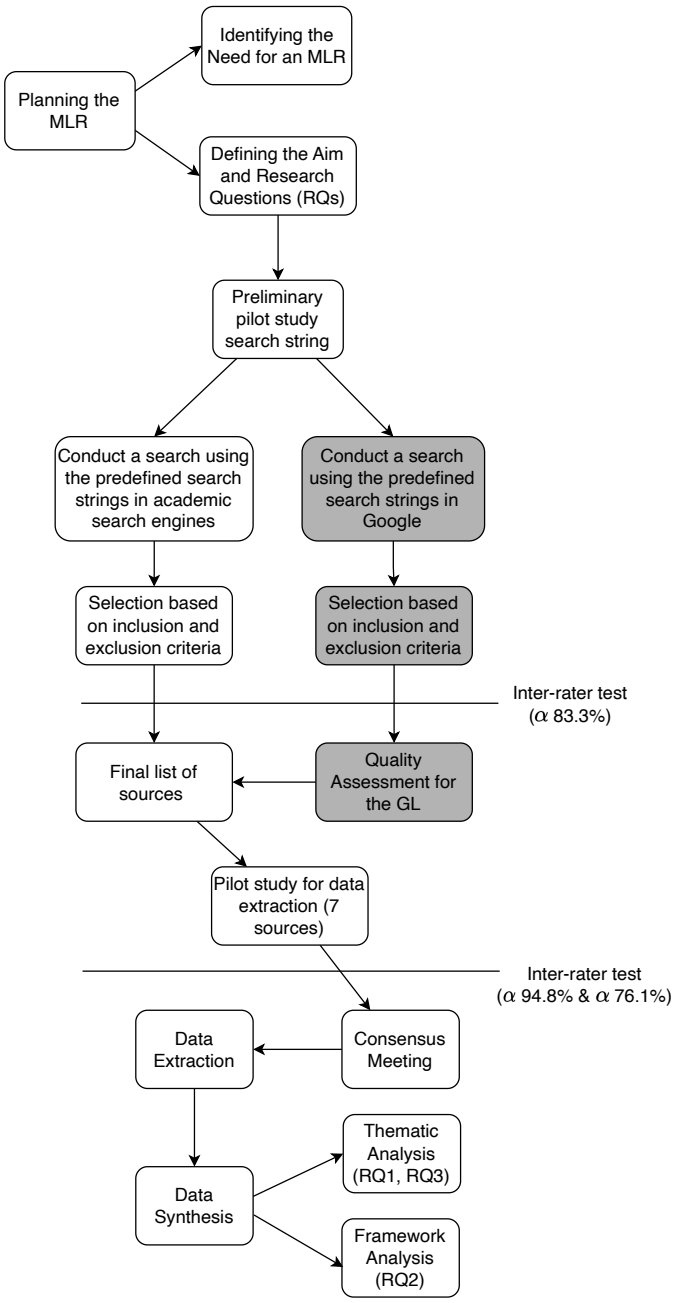

Figure 1: Outline for research methodology

3.1.1. Identifying the Need for an MLR

The MLR methodology was chosen over a traditional Systematic Literature Review (SLR) methodology due to the limited availability of peer-reviewed studies on smart contract upgradeability. MLRs integrate both peer-reviewed and grey literature (GL), providing a comprehensive understanding of emerging fields where industry practices often outpace academic research [32]. Grey literature, including technical blogs, white papers, tutorials, and community forum discussions, offers current and practical insights that may not yet be reflected in academic publications. By incorporating GL, we aim to capture both theoretical perspectives and real-world practices in smart contract upgrade mechanisms, ensuring a holistic view of the subject.

3.1.2. Defining the Aim and Research Questions (RQs)

The primary aim of this MLR is to explore, classify, and assess the methods for upgrading smart contracts, integrating perspectives from both academia and industry. The following research questions were formulated to address core aspects of smart contract upgradeability:

- **RQ1:** What are the existing approaches for upgrading Ethereum smart contracts? This question identifies and classifies the diverse methods and techniques used to upgrade smart contracts.
- **RQ2:** What are the characteristics of each upgrading approach? This question investigates the key characteristics of each approach based on core smart contract components, such as address preservation, logic upgrade scope, and storage management.
- **RQ3:** What are the benefits and limitations of each approach? This question evaluates the strengths and weaknesses of each approach based on software quality attributes such as flexibility, security, and scalability.

3.2. Search Strategy

The Search Strategy outlines the process of identifying key terms, conducting pilot testing, and developing the final search string to guide the literature search for this MLR.

3.2.1. Keyword Selection

We broke down the research questions into specific terms to develop a comprehensive set of keywords. Keywords were categorized into Ethereum-specific terms, upgrade-related terms, technical methods and patterns, and criteria to capture discussions on benefits and limitations. This structured approach to keyword selection ensured that both technical and practical aspects of smart contract upgradeability were thoroughly explored.

- **Ethereum-Specific Terms:** "Ethereum", "smart contract", "Solidity", "contract*"
- **Upgrade-Related Terms:** "upgrade", "update*", "mutability*", "immutability", "migration*"
- **Methods and Patterns:** "pattern", "architecture", "approach", "method", "proxy", "framework", "strategy"
- **Benefits and Limitations:** "security", "challenge*", "smell*", "issue*", "best practice", "benefit", "advantage*", "limitation*", "disadvantage*", "pro*", "con*"

The '*' wildcard is used to capture variations of a root term, ensuring the inclusion of related terms in the search results. For instance, "contract*" retrieves "contract", "contracts" and "contractual", while "challenge*" captures "challenge" and "challenges". Terms without the wildcard are already comprehensive and do not have meaningful variations relevant to the study. For example, "Ethereum" does not require a wildcard as there are no common variants.

3.2.2. Final Search String

We conducted pilot testing of the search string on a subset of databases and grey literature sources to assess its effectiveness. The pilot involved running the search string on IEEE Xplore and Google, reviewing the first 50 results to evaluate relevance. During pilot testing, we found that terms like "security" and "smell" yielded results related to general smart contract vulnerabilities rather than upgrade mechanisms. Therefore, these terms were removed to focus the search on upgradeability topics.

The following search string was developed to integrate the identified keywords, as well as the outcome of the pilot test:

```
("Ethereum" OR "smart contract*" OR "solidity" OR "contract*")
    AND
("upgrade*" OR "update*" OR "mutability*" OR "migration*") AND
("pattern" OR "approach" OR "method" OR "proxy" OR "strategy")
    AND
("issue" OR "challenge" OR "best practice" OR "benefit*" OR "
    advantage*" OR
"limitation*" OR "disadvantage*" OR "pro*" OR "con*")
```

The search string was adjusted as necessary to suit the syntax and indexing terms of each academic database. For grey literature searches on Google, parentheses were minimized to accommodate Google's search algorithms.

3.3. Source Selection

The Source Selection step outlines the approach taken to identify, include, and assess sources for this MLR, ensuring that only relevant academic and grey literature contributes to the analysis. Figure 2 provides an overview of the source selection process applied.

3.3.1. Grey Literature Sources

Grey literature was an essential source for capturing real-world practices. Searches were conducted through Google to include a diverse range of sources, such as blogs, archived white papers, videos, and community-based forums. The search was conducted using Google in incognito mode to prevent personalization bias, and the first 10 pages of results (approximately 100 results) were reviewed. The search was extended incrementally until theoretical saturation was achieved, following the guidelines proposed by Garousi et al. [32]. This ensured comprehensive coverage while maintaining search efficiency. Details of all collected grey literature sources from the initial search phase can be found in Appendix A.

3.3.2. Academic Literature Sources

To supplement the grey literature, academic databases including IEEE Xplore, SpringerLink, ACM Digital Library, and Scopus were searched. These databases were chosen for their extensive computer science and engineering coverage, particularly in blockchain and smart contract research. The refined search string was applied to each database, tailored to specific syntax requirements, and limited to English-language sources published from 2015 onwards (when Ethereum was introduced). To expand the search, backward

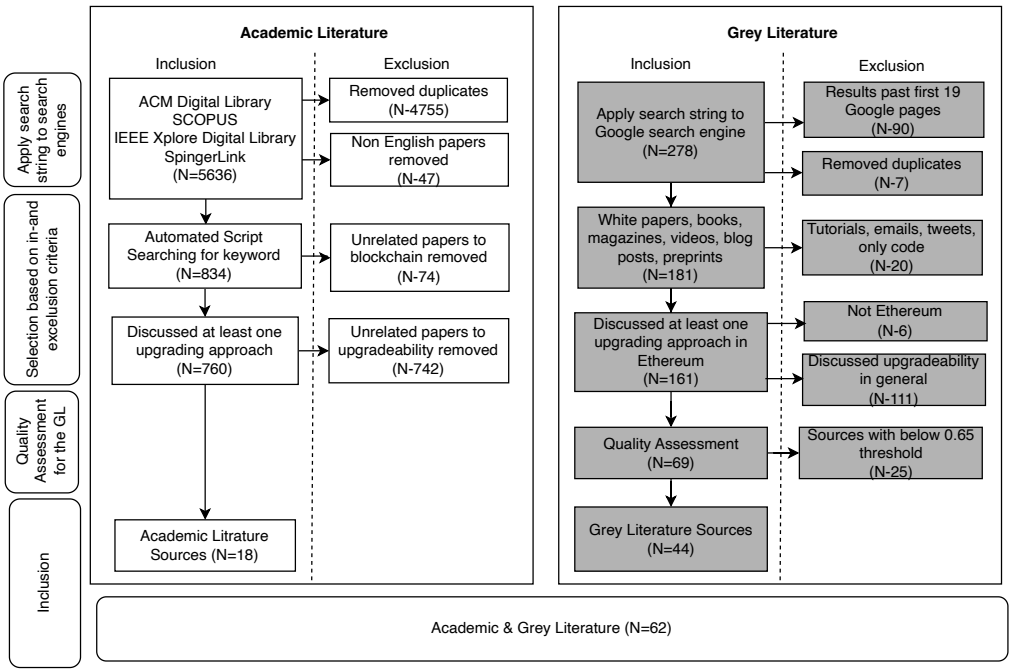

Figure 2: Overview of the source selection process for academic and grey literature

and forward citation tracking was performed on key articles. The results of this search are detailed in Appendix B.

3.3.3. Inclusion and Exclusion Process

To ensure rigorous selection of sources, predefined inclusion and exclusion criteria were applied to both grey literature and academic sources. For academic literature, automated scripts identified sources by scanning titles, abstracts, and keywords for terms such as Blockchain, Ethereum, Smart Contract, and Solidity. This process ensured a targeted focus on relevant studies in the blockchain domain, minimizing irrelevant data. The criteria applied during the source selection process are summarized in Table 1.

The screening procedure involved three independent reviewers who evaluated the titles, abstracts, and full texts of all identified sources. A majority rule decision approach was applied, where a source was included if at least two reviewers agreed on its relevance after the full-text review and excluded if at least two reviewers deemed it unsuitable. Discrepancies were resolved

Table 1: Inclusion and exclusion criteria applied for source selection

Inclusion Criteria	Exclusion Criteria
Sources on smart contract upgrading describing at least one pattern	Non-English sources Sources lacking technical depth (e.g., brief news articles, opinion pieces without analysis) Promotional content or advertisements Duplicated GLs Duplicate materials or summaries without new insights

through discussion to ensure consistency in the selection process. To assess the reliability of this screening, Krippendorff's Alpha was calculated, resulting in a score of 0.833, indicating high inter-rater reliability. Krippendorff's Alpha measures inter-rater agreement, accounting for chance and is suitable for various data levels, with scores above 0.80 considered strong [33]. Appendix C provides a detailed overview of the inclusion and exclusion decisions made during the selection process.

3.3.4. Quality Assessment

For grey literature, quality assessment was crucial, given the variability in source reliability. We adapted Garousi et al.'s quality assessment framework, covering 19 criteria grouped into seven quality categories: authority, methodology, objectivity, date, position with respect to related sources, novelty, impact, and outlet type. The first author independently assessed the quality of grey literature sources using the adapted framework. Each criterion was evaluated on a 3-point Likert scale (1 = Yes, 0.5 = Partly, 0 = No), and scores were averaged for a final quality score between 0 and 1.

After analyzing and calculating the quality scores for each source, it was essential to exclude low-quality sources to maintain the rigor and reliability of the review. In our case, we determined that any source with a score below 0.65 would be excluded. To establish this threshold, we employed an iterative quality threshold calibration process. We started with an initial high threshold of 0.8 and incrementally lowered it by 0.01, reviewing the sources that fell within each range. For each decrement, we assessed whether the sources added novel and valuable insights compared to those classified as high quality. This involved checking if the sources discussed broad patterns, provided limitations, or offered sufficient details that could be utilized, as opposed to merely listing information without enough depth. The process continued until we reached a threshold of 0.64, where sources below this score were

Table 2: Quality Categories and Criteria

Quality Category	**Quality Criteria**
Authority of the Producer	Is the publishing organization reputable? Is an individual author associated with a reputable organization? Has the author published other work in the field? Does the author have expertise in the area?
Methodology	Does the source have a clearly stated aim? Does the source have a stated methodology? Is the source supported by authoritative, contemporary references? Are any limits clearly stated? Does the work cover a specific question?
Objectivity	Is the statement in the source as objective as possible? Is there vested interest? Are the conclusions supported by data?
Date	Does the item have a clearly stated date?
Position w.r.t. Related Sources	Have key related GL or formal sources been linked to/discussed?
Novelty	Does it enrich or add something unique to the research? Does it strengthen or refute a current position?
Impact	Number of comments or views for specific online entries (blog posts, videos).
Outlet Type	1st Tier GL (measure = 1) 2nd Tier GL (measure = 0.5) 3rd Tier GL (measure = 0)

consistently of low quality and did not add significant new details. To ensure thorough evaluation, sources with scores as low as 0.6 were also reviewed to gain a comprehensive understanding of quality distribution. This method ensured that the final threshold of 0.65 balanced inclusiveness with quality, aligning with the recommendations by Garousi et al. [32] for maintaining robust criteria in MLRs. A detailed breakdown of the quality assessment for grey literature sources is provided in Appendix D.

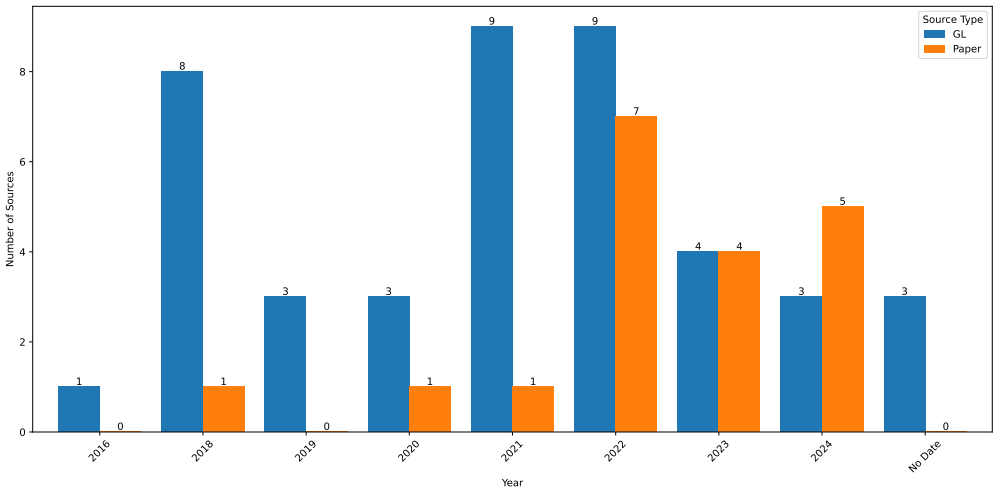

Figure 3: Distribution of included sources by publication year

3.3.5. Sources Statistics

A total of 62 sources were included in this study, comprising 19 academic papers and 43 grey literature (GL) sources, with data collected up to July 2024. The distribution of these sources by publication year is shown in Figure 3, where the x-axis represents the publication year and the y-axis indicates the number of sources. The figure highlights the increasing trend in publications over the years, demonstrating the growing interest and research in smart contract upgrade approaches. A comprehensive list of all included sources is provided in Appendix E.

3.4. Data Extraction Process

3.4.1. Pilot Phase

To refine the data extraction process, a pilot study was conducted with a selection of seven sources, which represented a mix of high and low-quality scores and included both grey and academic literature to test the extraction form's applicability across source types. A standardized data extraction form was developed to collect detailed descriptions of upgrade approaches, characteristics, and benefits or limitations linked to each approach, and notes to mention any extra information that does not fall in the predefined columns. Three reviewers independently extracted data from these sources and com-

pared results to check consistency, as detailed in Appendix F. Agreement rates were assessed using Krippendorff's Alpha, with the pilot showing high agreement for upgrade approaches (alpha = 0.948) and moderate agreement for characteristics and benefits/limitations (alpha = 0.761). Differences in interpretation were discussed, leading to revisions of the extraction form for clarity, including added definitions and clear definitions of each related aspect of the characteristics where the main differences were identified. The template used for data extraction is detailed in Appendix G.

3.4.2. Data Extraction

For the full data extraction, the same standardized form was used. Each reviewer independently extracted data from their assigned references, focusing on distinct sets of sources. To ensure consistency and address uncertainties, regular collaborative meetings were held throughout the extraction process. These discussions enabled reviewers to share insights, clarify ambiguities, and resolve uncertainties related to specific sources and approaches. A comprehensive summary of extracted data from all sources is provided in Appendix H.

3.5. Data Synthesis Process

In this study, we employed distinct data synthesis methodologies for each research question to systematically extract, analyze, and categorize the data from both academic and grey literature sources. These methodologies allowed us to provide a comprehensive answer to each research question while maintaining alignment with the structured approach of an MLR. Detailed descriptions of each method are provided in the corresponding sections and appendices.

1. RQ1: Classification of Smart Contract Upgrade Approaches

   To classify upgrade approaches, we conducted Thematic Analysis [17]. This method allowed us to identify recurring concepts and themes in upgradeability, facilitating the classification into Full Upgrade and Partial Upgrade Approaches. Collaborative Consensus Classification sessions ensured robust categorization and unified naming conventions. Further details are available in Section 5.

2. RQ2: Characteristics of Upgrade Approaches

For RQ2, we used Framework Analysis [18] to map each approach to its core components, address, logic, storage, and execution flow. This approach enabled us to categorize the upgrade methods based on characteristics such as address preservation, logic upgrade scope, storage management, and execution flow. Detailed descriptions can be found in Section 6.

3. RQ3: Benefits and Limitations of Upgrade Approaches

   We employed Thematic Analysis aligned with the ISO/IEC 25010 quality model to identify benefits and limitations. Each reviewer independently coded the benefits and limitations, followed by collaborative card-sorting sessions to group similar codes into broader themes such as complexity, flexibility, efficiency, security, and usability. Details of this process are provided in Section 7.

4. Systematization of Knowledge

Before addressing the research questions, it is essential to systematize the key concepts surrounding upgradeability in smart contracts. Understanding the fundamental components of smart contracts is crucial, as these elements directly influence discussions on upgradeability and its implications.

4.1. Fundamental Components of Smart Contracts

Smart contracts are composed of several fundamental components. The first is the **address**, a unique identifier on the blockchain through which users interact with the contract. The second is the **logic**, which defines the contract's functionality through its programmed rules, algorithms, and operations. The third is the **storage**, which maintains the contract's state, such as user balances or other persistent data. Lastly, the **execution flow** describes how transactions are processed, including the order of function calls and interactions within the contract.

These components are critical to understanding upgradeability because modifications to any of them can affect the contract's performance and the user experience. For instance, altering the logic may change the contract's functionality, while modifying storage can impact the integrity of the data maintained by the contract.

4.2. A Systematic Definition of Smart Contract Upgradeability

In the literature, there are differing views on what constitutes upgradeability in smart contracts, reflecting a tension between the need for continuity and the desire for flexibility in post-deployment contract changes. Some researchers and practitioners define upgradeability as the ability to preserve the state of the contract across upgrades, ensuring minimal disruption for users while maintaining access to their data and balances [34, 35, 36, 37, 13, 14]. This perspective is particularly emphasized in contexts like decentralized finance (DeFi), where user continuity and trust are critical. Maintaining a seamless experience for users without requiring manual action during upgrades is often highlighted as a fundamental characteristic of an upgradeable smart contract.

Conversely, others consider approaches where state preservation is unnecessary for upgradeability [38, 12, 39]. In this view, deploying a new contract and requiring users to migrate data manually can still be classified as an upgradeable solution if it enables modifications to the contract logic, such as allowing new functionalities or significant updates. This interpretation broadens the scope of upgradeability to approaches that enable logical modifications, even if users must take extra steps to reconnect with the contract.

Additionally, perspectives differ on the level of flexibility necessary for upgradeability, particularly concerning the preservation of the interaction address. Some literature emphasizes approaches that allow modifications to any contract component while maintaining the original interaction address, ensuring seamless user interaction and adaptability for substantial system updates [40, 41, 42]. Others discuss more open-ended approaches that do not maintain the original interaction address, allowing extensive changes to components and functionalities. This flexibility can result in users needing to interact with a new address after updates, thereby sacrificing consistency in user interaction [38, 12, 39]. While these more open approaches support significant modifications, they may impact user trust and continuity.

Given these differing views, there is a clear need to develop a unified definition of upgradeability that accommodates these perspectives while recognizing the unique constraints of smart contracts. Drawing from standard definitions of software upgradeability and considering the structure of smart contracts, we propose the following definition:

Upgradeability in smart contracts refers to the capability to modify the contract's logic or code post-deployment. This modification can occur with or

without preserving the contract's state and may involve full or partial changes to the contract's components.

This definition acknowledges that upgradeability can be achieved through various means, including both state-preserving and non-state-preserving approaches, and that it can involve modifications to different contract components. It emphasizes the importance of predefined mechanisms within the contract that allow for upgrades, distinguishing these from external interventions that may undermine the contract's autonomy.

5. RO1: Smart Contract Upgrading Approaches

To address RQ1, which aims to classify existing smart contract upgrade methods, we employed Thematic Analysis as our data extraction and synthesis method [17]. This approach allowed us to systematically identify, analyze, and report patterns within our collected data from both academic and grey literature sources. We began by thoroughly familiarizing ourselves with the literature to generate initial codes based on recurring concepts related to upgradeability. Guided by the unified definition of upgradeability established in Section 4.2, these codes were organized into potential themes, leading to the classification of upgrade methods into two primary categories: Full Upgrade Approaches and Partial Upgrade Approaches (discussed in detail in Section 5.1).

This classification highlights the fundamental differences in how upgrades impact a contract's state and user interactions, providing a framework for evaluating existing approaches. To ensure the robustness of this classification, we conducted Collaborative Consensus Classification sessions among the three authors (Author 1, Author 2, and Author 3). During these discussions, we reviewed and deliberated on the identified approaches to reach an agreement on their categorization. Recognizing the varied terminologies used across sources, we also unified the naming conventions to standardize the terminology (see Table 3 in Section 5.2). This table showcases the chosen unified names alongside the alternative names found in the literature, addressing potential ambiguities and improving clarity in communication. Through iterative refinement, we ensured that our classifications accurately represented the data and aligned with our unified definition of upgradeability. A detailed description of the Thematic Analysis process for RQ1 is provided in Appendix I.

5.1. Classification of Smart Contract Upgrading Approaches

The classification of smart contract upgrade methods is organized into two main categories: Full Upgrade Approaches and Partial Upgrade Approaches. This structure captures the differences in how each method modifies contract logic and storage, providing a clear framework for understanding the variety of upgrade techniques identified through thematic analysis.

5.1.1. Full Upgrade Approaches

In Full Upgrade approaches, the entire smart contract is redeployed when an upgrade is required. This process involves replacing both the contract's logic and state, resulting in a completely new contract version.

- **Contract Migration:** Involves deploying a new smart contract at a different address to replace an existing one. The state, including balances and variables, is manually migrated from the old contract to the new one. Users must be informed of the change and update their interactions to the new contract address as the old contract becomes obsolete. This approach requires careful handling of state migration to avoid inconsistencies and may disrupt interactions with the system during the migration process. Developers often make official announcements to facilitate the transition, and platforms like Etherscan may label the old contract as "Old Contract" to notify users.
- **Metamorphic Contracts (CREATE2):** This upgrade mechanism leverages the CREATE2 opcode to redeploy a contract at the same address after the original contract self-destructs. Developers use `SELF DESTRUCT` to remove the existing contract's code and state, freeing up the address. A new contract with updated logic is then deployed to the same address using CREATE2, which allows pre-determination of the contract's address based on a hash of the deploying account, salt, and initialization code. While the contract's address remains the same, the state is not preserved, as using `SELFDESTRUCT` clears the state.

5.1.2. Partial Upgrade Approaches

Partial Upgrade approaches are designed to modify the smart contract's logic while preserving the contract's state. The two primary subcategories of partial upgrades are Two-Module Approaches and Multi-Module Approaches.

Two-Module Approaches. Two-module approaches separate a contract's logic and state into two distinct modules, which allows for the full upgrade of the logic component through new deployments while the state remains stable. Two primary techniques within this category include Data Separation and Proxy-Based Approaches.

- **Data Separation:** In this approach, a contract is divided into a logic contract that manages operations and a data contract that preserves the contract's state. Users interact with the logic contract, which references the data contract using opcodes like `CALL` or `STATICCALL` to access or modify data. To secure the state, the data contract restricts updates to those initiated by the designated logic contract address.

  For upgrades, developers deploy a new logic contract (at a new address) and update this address within the data contract's storage. This change requires users to interact with the new address, so developers often employ a registry that maintains the latest logic contract address. This allows users to retrieve the current address and interact seamlessly with the latest contract version. There are three main types of data separation: Inherited Storage, Eternal Storage, and Unstructured Storage.

  - **Inherited Storage:** Maintains a consistent storage layout across upgrades using Solidity's inheritance mechanism. A separate Storage Contract defines all the state variables used by the contract. The logic contract and any upgraded versions inherit from this Storage Contract, ensuring they share the same storage structure. Developers must ensure the storage layout remains consistent, avoiding changes to existing state variables to prevent misalignment.
  - **Eternal Storage:** Abstracts data storage into a separate contract using key-value pairs, typically implemented through mappings. The logic contract interacts with this storage contract using setters and getters to read or modify data. This allows developers to modify the logic contract freely without worrying about storage layout or data corruption.
  - **Unstructured Storage:** Organizes data within specific storage slots identified by unique hashes, such as those generated by

`keccak256`. The logic contract directly accesses and modifies these slots, often using low-level assembly code. This method eliminates the need for a predefined storage layout, optimizing data access efficiency.

- **Proxy-Based Approaches:** Use a proxy contract to delegate calls to a logic contract while maintaining state within the proxy.
  - **Basic Proxy:** The simplest form of the proxy pattern, holding state, and forwarding function calls to the logic contract using `delegatecall`. Users interact with the proxy, which maintains a consistent address. Upgrades involve deploying a new logic contract and updating the proxy's reference.
  - **EIP-897 (Delegate Proxy):** Distinguishes between different proxy use cases, defining strict and upgradable proxies. It introduces interfaces that signal the proxy's purpose and primary implementation.
  - **EIP-1967 (Standard Storage Slots):** Defines storage slots for the logic contract's address and upgrade-related variables. Upgrades involve updating the storage slot pointing to the logic contract.
  - **UUPS (Universal Upgradeable Proxy Standard):** The proxy contract delegates calls to the logic contract but does not contain the upgrade mechanism. The logic contract includes the upgrade functionality.
  - **Transparent Proxy:** Differentiates between admin and user functions, handling admin-only functions directly while forwarding user calls to the logic contract.
  - **Beacon Proxy:** Uses a beacon contract to store the logic contract's address. Proxies retrieve the address from the beacon, allowing for simultaneous upgrades of multiple proxies.

Multi-Module Approaches. Multi-module approaches divide the contract's logic into multiple modules, each upgradable independently.

- **Strategy Pattern:** Delegates logic to external strategy contracts, allowing targeted upgrades without altering the main contract.

- **EIP-1538 (Transparent Contract Standard):** Allows dynamic proxy-based upgrades, adding, replacing, or removing functions at runtime.
- **Diamond Pattern (EIP-2535):** Divides the contract into facets, each handling specific functions, enabling modular upgrades.

Hybrid Approaches. Combine elements of data separation and proxy mechanisms to ensure storage compatibility.

- **Proxy with Inherited Storage:** Aligns with Inherited Storage, where the logic contract inherits storage layout from the proxy.
- **Proxy with Eternal Storage:** Uses a separate storage contract, allowing logic contract upgrades without affecting the state.
- **Proxy with Unstructured Storage:** Utilizes predefined storage slots for variables the logic contract manages.

5.2. Unified Smart Contract Upgrading Terminologies

Throughout our literature review, we encountered various terms used across sources to describe the same upgrade approaches, often creating ambiguity and making clear communication challenging. For instance, what we refer to as "Basic Proxy" is frequently labeled as "Proxy Contract," a term broadly applied to several proxy types. Similarly, "Data Segregation" is used variably, sometimes encompassing both what we classify as Data Separation and the Strategy Pattern, adding further confusion. Additionally, the term "Transparent" is inconsistently used to describe both EIP-1538 and Transparent Proxy approaches. We have adopted a unified terminology throughout our classification to address these inconsistencies for consistency and clarity. In Appendix L, we provide a comprehensive table mapping various names in the literature to our chosen standardized terms. This table serves as a reference, allowing readers to cross-reference and understand the relationships between different terms used in the field. Table 3 shows a sample of this table.

Table 3: Standard Terminologies for Upgrading Approaches

Unified Term	Alternative Names
Contract Migration	Basic Contract Upgrade, Social Migration
Data Separation	Data separation pattern, Separate Logic Contract, Basic Data Segregation
Basic Proxy	Delegatecall-based proxies pattern, Proxy contract
Strategy Pattern	Updates through functions, Partially Upgradeable Contract Systems

5.3. Statistics on Upgradeability Approaches

Figure 4 presents a stacked bar chart depicting the distribution of mentions for different smart contract upgrade approaches across grey literature (GL) and academic papers, highlighting which techniques are most commonly discussed in each source type. Results show that Basic Proxy is the most frequently mentioned approach, aligning with the broadly accepted definition of upgradeability, which emphasizes preserving contract state across upgrades. Although other approaches, such as Data Separation, also maintain state continuity, proxies, especially Basic Proxy, dominate the conversation. Interestingly, a few studies (16.13%) mention the Registry Pattern as an upgradeable mechanism; however, we do not consider it an upgrade approach per our definition, as it merely stores the latest contract version and can be used with other approaches. Additionally, proxy variations receive broad coverage in grey literature, while academic papers show more selective emphasis, especially on hybrid patterns such as Proxy with Eternal Storage. Lastly, data separation techniques, such as Eternal Storage, are largely discussed within grey literature, with only one academic mention [43], illustrating differing emphases across literature types.

5.4. RQ1 Answer

To address **RQ1: What are the existing approaches for upgrading Ethereum smart contracts?**, we conducted a thematic analysis, identifying 17 distinct upgrade approaches. The classification of smart contract upgrades is structured around the scope of modifications(logic and storage). Figure 5 provides an overview of the proposed classification, dividing upgrade methods into two main categories, Full Upgrade Approaches, and Partial Upgrade Approaches, and detailing the specific methods identified through thematic analysis.

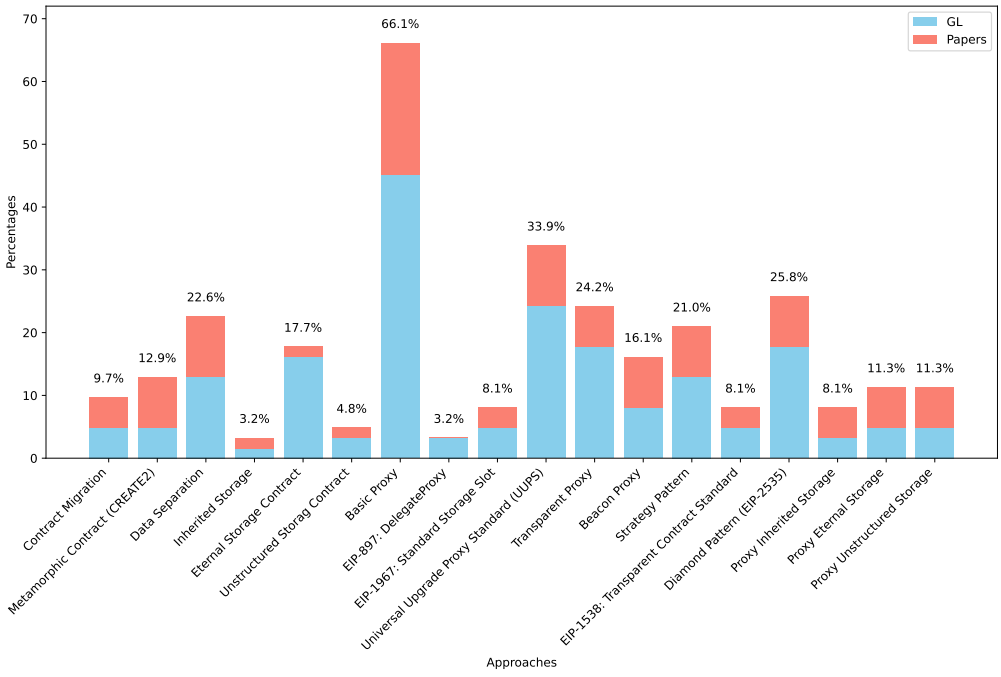

Figure 4: Percentage of mentions for smart contract upgrade approaches across grey literature (GL) and academic papers

6. RO2: Characteristics of Smart Contract Upgrading Approaches

In this section, we address RQ2 by discussing and classifying the characteristics of various smart contract upgrade approaches, defined in RQ1, based on the core components of smart contracts introduced earlier (Section 4.1). To systematically analyze how each upgrade approach interacts with these components (address, logic, storage, and execution flow), we employed Framework Analysis as our data synthesis method [18]. This method is particularly suitable for applied research with specific questions and predefined issues, allowing us to organize and interpret data within a structured framework.

We began by mapping each upgrade approach to the core components, examining the specific ways in which they affect or utilize each component. This process involved familiarization with the data, identifying key themes related to each component, and indexing the data accordingly. We then

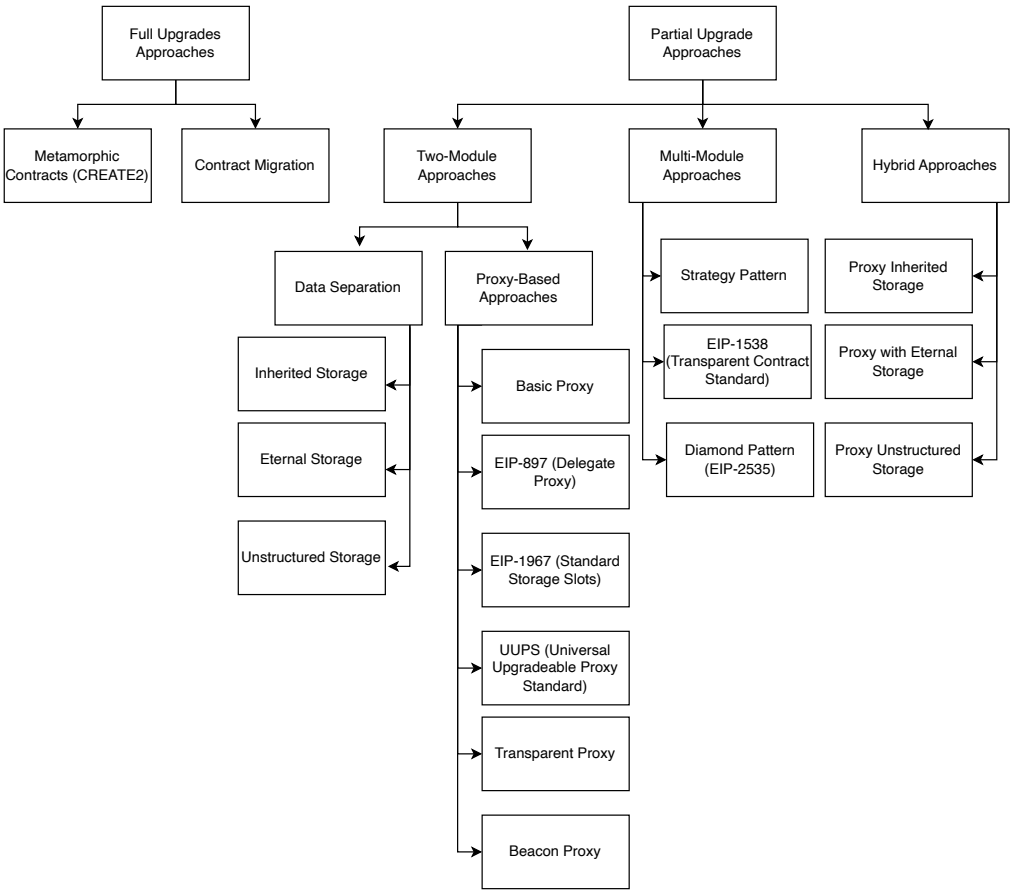

Figure 5: Overview of the classification of smart contract upgrade approaches

charted the data into a matrix format, facilitating a clear comparison across different approaches.

By using Framework Analysis, we were able to categorize the upgrade approaches based on characteristics such as address preservation, logic upgrade scope, storage management, and execution flow. A detailed explanation of the Framework Analysis process for RQ2 is provided in Appendix J.

6.1. Address Preservation

Address preservation determines whether the contract's address, which users interact with, remains consistent during an upgrade. This component

has significant implications for user experience and interaction continuity. There are two types:

- Fixed Address: These approaches ensure that users continue interacting with the same contract address even after an upgrade, enhancing trust and reducing the need for changes in user behavior. Proxy-based approaches (Basic Proxy, UUPS, EIP-1967, Transparent Proxy, Beacon Proxy), Multi-Module Approaches (Diamond Pattern, EIP-1538, Strategy Pattern), and Hybrid Approaches (Proxy Inherited Storage, Proxy with Eternal Storage, Proxy Unstructured Storage) fall under this type. Metamorphic Contracts (CREATE2) also maintain a fixed address, providing predictable user interactions.

- Changing Address: In these approaches, users interact with a new address every time there is an upgrade, leading to a fragmented experience and the need for users to update their references. This is characteristic of Contract Migration and Data Separation approaches (Inherited Storage, Eternal Storage, Unstructured Storage), where a new version of the contract is deployed at a different address.

6.2. Logic Upgrade Scope

The scope of a logic upgrade defines how extensively the logic of a contract can be modified during an upgrade, which impacts the system's flexibility. There are two types:

- Full Logic Replacement: Approaches that allow complete replacement of the contract's logic to enable substantial changes. This is common in Full Upgrade Approaches (Contract Migration, Metamorphic Contracts), Proxy-Based Approaches (Basic Proxy, UUPS, EIP-1967, Transparent Proxy, Beacon Proxy), Hybrid Approaches (Proxy Inherited Storage, Proxy with Eternal Storage, Proxy Unstructured Storage), and Two-Module Approaches like Data Separation (Inherited Storage, Eternal Storage, Unstructured Storage).

- Partial Logic Replacement: Approaches that support updating specific logic modules without changing the entire contract. This type is seen in Multi-Module Approaches (Diamond Pattern, EIP-1538, Strategy Pattern), which offer targeted upgrades and modularity.

6.3. Storage Management

Storage management addresses how a contract's state is maintained during upgrades, which affects data integrity and continuity. This component has two types:

- State Preservation: These approaches keep the contract's state intact during upgrades, maintaining data continuity and minimizing disruption. All approaches in the partial upgrade category in RQ1 (Two-Modules, Multi-Module, and Hybrid approaches) fall into this category.
- State Migration Required: In these approaches, the contract's state must be transferred when a new version is deployed, introducing complexities and potential risks. Contract Migration and Metamorphic Contracts (CREATE2) typically require state migration.

6.4. Execution Flow

Execution flow refers to how function calls are processed within the contract and whether they are direct or delegated. This characteristic affects the modularity and performance of contract upgrades. There are two types:

- Delegate Calls: Approaches that use delegatecall to execute logic in the context of the proxy's storage, facilitating logic upgrades while keeping the state consistent. Proxy-based approaches (Basic Proxy, UUPS, EIP-1967, Transparent Proxy, Beacon Proxy), Multi-Module Approaches, except Strategy pattern, and Hybrid Approaches use delegate calls.
- Direct Calls: Approaches that execute function calls directly in the contract, simplifying the process but limiting modularity. Contract Migration, Metamorphic Contracts (CREATE2), Data separation approaches, and the Strategy Pattern within Multi-Module Approaches use direct calls.

6.5. RQ2 Answer

To answer RQ2: What are the characteristics of each upgrading approach?, we conducted a framework analysis of the approaches identified in RQ1. Table 4 provides an outline of the characteristics of smart contract upgrade approaches, highlighting how each approach manages address preservation, logic upgrade scope, storage management, and execution flow.

Table 4: Characteristics of Smart Contract Upgrade Approaches Based on Core Components

Approach	Address Preservation	Logic Upgrade Scope	Storage Management	Execution Flow
Contract Migration	Changing Address	Full Replacement	State Migration Required	Direct Calls
Metamorphic Contracts (CREATE2)	Fixed Address	Full Replacement	State Migration Required	Direct Calls
Inherited Storage	Changing Address	Full Replacement	State Preservation	Delegate Calls
Eternal Storage	Changing Address	Full Replacement	State Preservation	Delegate Calls
Unstructured Storage	Changing Address	Full Replacement	State Preservation	Delegate Calls
Basic Proxy	Fixed Address	Full Replacement	State Preservation	Delegate Calls
UUPS	Fixed Address	Full Replacement	State Preservation	Delegate Calls
EIP-1967	Fixed Address	Full Replacement	State Preservation	Delegate Calls
Transparent Proxy	Fixed Address	Full Replacement	State Preservation	Delegate Calls
Beacon Proxy	Fixed Address	Full Replacement	State Preservation	Delegate Calls
Diamond Pattern (EIP-2535)	Fixed Address	Partial Replacement	State Preservation	Delegate Calls
EIP-1538	Fixed Address	Partial Replacement	State Preservation	Delegate Calls
Strategy Pattern	Fixed Address	Partial Replacement	State Preservation	Direct Calls
Proxy Inherited Storage	Fixed Address	Full Replacement	State Preservation	Delegate Calls
Proxy with Eternal Storage	Fixed Address	Full Replacement	State Preservation	Delegate Calls
Proxy Unstructured Storage	Fixed Address	Full Replacement	State Preservation	Delegate Calls

7. RO3: Evaluation of Smart Contract Upgrade Approaches

To systematically identify and categorize the benefits and limitations of each smart contract upgrade approach, we conducted a thematic analysis as part of our Multivocal Literature Review (MLR). Each reviewer independently coded the data, highlighting key benefits and limitations mentioned in the literature.

We then held collaborative sessions to discuss our findings, using card-sorting techniques to group similar codes into broader themes. This process allowed us to identify common evaluation criteria such as Complexity, Flexibility, Efficiency, Security, and Usability. We aligned these themes with recognized software quality models, particularly the ISO/IEC 25010 standard, adapting them to the specific context of smart contracts. In the following sections, we evaluate each upgrade approach against these metrics, considering the components above and comparing their benefits and limitations. Details of the thematic analysis process for RQ3 are provided in Appendix K.

7.1. Complexity

Complexity is crucial in evaluating smart contract upgrade approaches, as it impacts the effort needed to modify, maintain, or upgrade the system. According to ISO/IEC 25010, complexity, which corresponds to maintainability, is defined as:

The degree of effectiveness and efficiency with which the intended maintainers can modify a product or system.

We assess complexity across three key aspects of smart contract components: logic, storage, and execution flow. We did not include the address component, as it does not influence the complexity.

7.1.1. Logic Complexity

Logic complexity refers to the challenge of updating or modifying a contract's logic without introducing errors. We classify logic complexity into four levels based on difficulty, where Low indicates ease of modification and Very High reflects significant difficulty.

- **Low Logic Complexity:**
  - *Contract Migration:* Allows developers to rewrite the entire contract logic without needing to consider compatibility with previous versions, making the process straightforward and manageable [38, 37, 12, 44].
  - *Strategy Pattern:* Enables isolated updates to specific functionalities without affecting the core contract, reducing the overall complexity of logic modifications [45, 39, 46, 12].
- **Moderate Logic Complexity:**
  - *Data Separation Patterns (Inherited Storage, Eternal Storage, Unstructured Storage):* Maintain separate logic and storage, simplifying upgrades but requiring some coordination to ensure integration, resulting in moderate complexity [38, 47, 48, 49, 12].
  - *UUPS Proxy:* Embeds the upgrade function directly within the logic contract, streamlining upgrades, though it requires caution to avoid issues like bricking [34, 50].
  - *Metamorphic Contracts (CREATE2):* Support full rewrites of contract logic without complex structures, but redeployment steps demand attention to maintain the desired setup [34, 12, 51, 52, 53].
- **High Logic Complexity:**
  - *Proxy-Based Approaches (Basic Proxy, Transparent Proxy, Beacon Proxy, EIP-897, EIP-1967):* Rely heavily on `delegatecall`, which increases complexity due to the need for precise function selector management and consistent mapping to avoid clashes [48, 54, 55, 56, 14].

  - *Hybrid Approaches (Proxy with Eternal Storage, Proxy Inherited Storage, Proxy Unstructured Storage):* Combine proxy functionality with specific storage handling, requiring careful control of storage pointers and proxy forwarding, adding significant complexity [45, 57, 15, 34].

- **Very High Logic Complexity:**
  - *Diamond Pattern (EIP-2535):* The facet-based structure allows extensive modularity but requires precise management of function selectors and alignment across facets to avoid misalignment and unintended interactions [38, 34, 14, 34, 41, 51, 58, 59].
  - *EIP-1538 (Transparent Contract Standard):* Supports dynamic addition and removal of functions, which demands ongoing adjustments in function mapping and careful oversight to prevent clashes and logic errors [51, 52, 60].

7.1.2. Storage Complexity

Storage Complexity refers to the effort to manage or maintain the contract's state during upgrades. We assess storage complexity based on difficulty, where Low indicates straightforward state management, and Very High signifies significant challenges.

- **Low Storage Complexity:**
  - *Proxy-Based Approaches:* This includes Basic Proxy, Transparent Proxy, Beacon Proxy, EIP-897, UUPS Proxy, and EIP-1967 Standard Storage Slots. These approaches maintain state within the proxy itself, avoiding the need for state migration or extensive coordination, making storage management straightforward [48, 61, 12, 50].
  - *Strategy Pattern:* Isolates state within the core contract, with individual strategies operating independently without affecting core storage. This simplicity results in a low level of storage complexity [45, 39, 46, 12, 53].

- **Moderate Storage Complexity:**
  - *Data Separation Patterns:*

    - *Unstructured Storage:* Flexible mappings and hash-based keys simplify storage management without strict adherence to predefined structures [62, 13].
    - *Inherited Storage:* Provides consistent storage layout across contracts, requiring careful but not overly complex alignment [38, 47, 49, 12, 11].
  - *Hybrid Approaches (Proxy with Inherited Storage, Proxy Unstructured Storage):* Combine proxy mechanisms with simpler storage contracts, adding moderate complexity through pointer management and ensuring storage consistency across contracts [45, 57, 15].

- **High Storage Complexity:**
  - *Data Separation - Eternal Storage:* Involves complex storage access due to its abstract structure. Managing data types like structs and mappings increases the complexity of updates and alignment, making it more challenging than other Data Separation types [63, 49, 64, 43].
  - *Hybrid Approaches (Proxy with Eternal Storage):* Eternal Storage within a hybrid setup requires detailed management of storage mappings and state consistency, adding high storage complexity [45, 57, 15].
  - *Diamond Pattern (EIP-2535):* The modular structure requires shared storage among facets, necessitating careful layout management to prevent collisions and ensuring that facets interact correctly with shared storage [38, 34, 14, 34, 41, 51, 58, 59].
  - *EIP-1538 (Transparent Contract Standard):* Similar to Diamond, its dynamic function modifications require consistent storage updates across modules, adding to storage management demands [51, 52, 60, 14].

- **Very High Storage Complexity:**
  - *Contract Migration:* Involves full state migration to a new contract, requiring extensive manual data transfer and state alignment, resulting in significant resource and time costs [38, 37, 12, 44].

– *Metamorphic Contracts (CREATE2):* Lack state preservation post-self-destruction, necessitating external management for any retained data. Reinitializing the state upon redeployment leads to very high storage complexity for continuity [34, 12, 39, 52, 65].

7.1.3. Execution Flow Complexity

Execution Flow Complexity refers to the complexity introduced in the contract's execution flow due to the upgrade mechanism. Similar to the previous components, we rank this complexity based on difficulty, where Low means minimal changes and Very High means major challenges.

- **Low Execution Flow Complexity:**
  - *Contract Migration:* Simple execution flow with no complex delegation or routing requirements. Users interact directly with the new contract, and upgrades involve full redeployment rather than layered routing.
  - *Strategy Pattern:* The execution flow remains straightforward since each strategy operates independently, allowing the core contract to delegate calls to specific strategies without impacting the main execution structure [38, 37].
  - *Metamorphic Contracts (CREATE2):* Execution remains simple as each upgrade involves redeployment rather than intricate call routing or storage dependencies [39, 52, 65].
- **Moderate Execution Flow Complexity:**
  - *Proxy-Based Approaches (including Basic Proxy, UUPS Proxy, EIP-897, and EIP-1967):* These proxies add a moderate level of complexity by using `delegatecall` to route calls to the logic contract. However, they do not require additional role-based distinctions or external dependencies [48, 66, 67, 38, 53, 14].
    * *Basic Proxy:* Manages straightforward `delegatecall` routing to the logic contract.
    * *UUPS Proxy:* Stores the upgrade function within the logic contract itself, requiring fewer admin roles but still necessitating `delegatecall` management during upgrades.

    - *EIP-897 and EIP-1967:* Implement similar `delegatecall` routing while avoiding complex role management.
  - *Data Separation Patterns (Unstructured Storage, Inherited Storage):* While they involve an additional call to access separate storage, their execution flow remains relatively simple. The logic and storage layers are distinct, but no significant routing complexity is added beyond storage retrieval [38, 47, 48, 49, 12].

- **High Execution Flow Complexity:**
  - *Transparent Proxy:* Requires additional role management to separate admin from user functions, creating layers in the execution flow and increasing complexity, especially when managing permissions for upgrades [50, 36].
  - *Beacon Proxy:* Adds an external dependency on a centralized beacon contract for logic contract addresses, adding a routing step and reliance on the beacon for directing proxies [68, 69, 45].
  - *Hybrid Approaches (Proxy with Eternal Storage, Proxy Inherited Storage, Proxy Unstructured Storage):* Combine proxy routing with specific storage contracts, introducing higher execution flow complexity due to the additional storage contract coordination and external calls [45, 57, 15, 70].

- **Very High Execution Flow Complexity:**
  - *Diamond Pattern (EIP-2535)* and *EIP-1538 (Transparent Contract Standard):* These introduce very high execution flow complexity, as managing multiple facets or dynamically adding and removing functions increases both the routing complexity and the risk of execution path mismanagement [59, 60, 12, 51].

7.1.4. Summary of Complexity Rankings

The heatmap, presented in Figure 6, provides a comprehensive visual summary of the complexity levels across various smart contract upgrade approaches. The X-axis represents the three aspects of complexity (Logic Complexity, Storage Complexity, and Execution Flow Complexity), while the Y-axis lists each smart contract upgrade approach individually. Color coding is utilized to indicate complexity levels, ranging from low to very

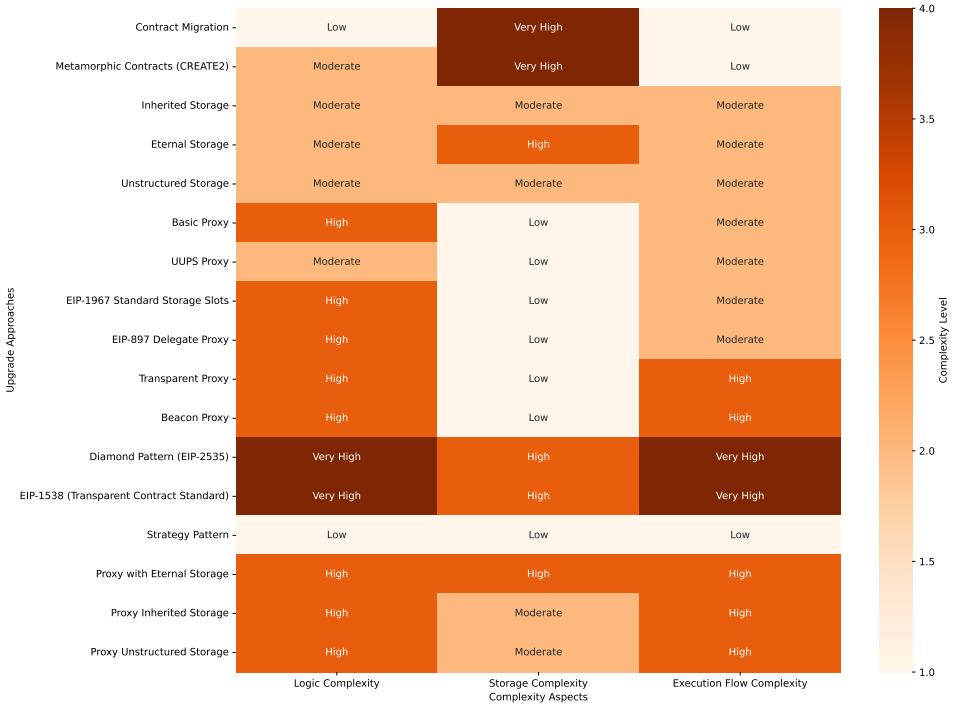

Figure 6: Complexity levels for smart contract upgrade approaches

high. Darker colors signify higher complexity, while lighter shades represent lower complexity.

We calculated an average complexity score for each approach across the three aspects to classify and interpret the complexity data. Complexity scores were assigned based on a 1-4 scale, where 1 is low complexity, and 4 is very high complexity. Using these scores, we derived an overall complexity classification for each approach as follows:

- **Low Complexity:**
  - *Strategy Pattern*: Exhibits the lowest complexity among all approaches, scoring 1. Its isolated, core-focused structure enables upgrades to specific functionalities without affecting the primary execution or storage setup.
  - *Contract Migration*: Also falls into the low complexity category,

allowing developers to rewrite the entire contract logic without considering compatibility with previous versions. This straightforward process makes it manageable and efficient.

- **Moderate Complexity:**
  - *Proxy-Based Approaches*: Including Basic Proxy, Transparent Proxy, Beacon Proxy, UUPS Proxy, EIP-897, and EIP-1967, maintain moderate complexity levels. These approaches streamline upgrades but require careful function management due to the use of `delegatecall`, which adds layers of complexity to both the logic and execution flow.
  - *Data Separation Patterns*: Such as Inherited Storage, Eternal Storage, and Unstructured Storage exhibit moderate complexity as well. While these patterns allow for separate management of logic and storage, they still require coordination to ensure integration.
  - *Metamorphic Contracts (CREATE2)*: Also fit into this category due to their capacity for full logic rewrites, which simplifies some aspects of complexity but introduces challenges regarding state preservation.
- **High Complexity:**
  - *Hybrid Approaches*: Which include Proxy with Eternal Storage, Proxy Inherited Storage, and Proxy Unstructured Storage, are classified as high complexity due to their combination of proxy functionalities with specific storage management. These approaches demand careful coordination between the proxy and storage layers, increasing the overall complexity.
- **Very High Complexity:**
  - *Diamond Pattern (EIP-2535)* and *EIP-1538 (Transparent Contract Standard)*: Fall into the very high complexity category. Both approaches involve extensive modularity and dynamic function management, necessitating careful management of function selectors and alignment across facets to avoid unintended interactions.

7.2. Flexibility

Flexibility is an important factor in smart contract upgrade approaches, as it indicates how easily a contract can be adapted to meet changing requirements. According to ISO/IEC 25010, adaptability is defined as:

The degree to which a product or system can be effectively and efficiently adapted for different or evolving hardware, software, or other operational or usage environments.

We evaluate flexibility in terms of logic and storage, as these components significantly impact how adaptable a contract is. Execution flow and address are not included in this evaluation because they do not directly influence the contract's adaptability.

7.2.1. Logic Flexibility

Logic Flexibility refers to how easily a smart contract's logic can be updated, extended, or modified to add new features or functionality without significant limitations. We rank logic flexibility in four levels, where low indicates limited adaptability and very high means extensive adaptability and ease of modification.

- **Low Logic Flexibility:**
  - *Contract Migration*: Updating individual functions is not possible, as each change requires deploying a completely new contract, limiting flexibility to refine or expand existing logic incrementally [38, 37, 12, 44].
  - *Metamorphic Contracts (CREATE2)*: Although redeployment is possible at the same address, the entire logic must be replaced each time, as isolated updates to specific functions are unsupported [51, 45].
- **Moderate Logic Flexibility:**
  - *Inherited Storage*: New functions can be introduced through inheritance, but modifying inherited logic poses challenges, limiting adaptability for direct updates to existing functions [34, 47, 49, 13].
  - *Strategy Pattern*: Functionality within specific areas can be updated by swapping strategy contracts, allowing isolated changes

without altering the primary logic, but broader updates are limited [15, 38, 71].

- **High Logic Flexibility:**
  - *Data Separation Patterns (Eternal Storage, Unstructured Storage)*: With storage managed separately, new functions or modifications can be introduced to the logic without affecting stored data, supporting flexibility in evolving logic [63, 49, 13, 62].
  - *Proxy-Based Approaches (Basic Proxy, Transparent Proxy, Beacon Proxy, EIP-897, EIP-1967, UUPS)*: These proxies allow the replacement of the implementation contract, supporting the addition of new functions or modifications to existing ones within the constraints of the proxy structure [48, 55, 34, 38, 12, 69].
- **Very High Logic Flexibility:**
  - *Diamond Pattern (EIP-2535)*: Facet-based structure enables modular updates, allowing functions to be added, replaced, or removed independently within facets, offering highly adaptable logic [13, 72, 12, 59, 14].
  - *EIP-1538 (Transparent Contract Standard)*: Supports on-the-fly updates to add, replace, or remove functions dynamically, providing unrestricted flexibility for granular modifications to contract functionality [59, 60, 12, 51, 14].

7.2.2. Storage Flexibility

Storage Flexibility refers to the ability to modify storage structures, such as adding new state variables or altering existing data without causing conflicts or inconsistencies. Similar to logic flexibility, we rank storage flexibility into four levels, where Low indicates limited adaptability and Very High represents extensive adaptability and ease of modification.

- **Low Storage Flexibility:**
  - *Contract Migration*: Storage is reset during migration, and the layout is fixed; updates to existing data structures or the addition of new variables are restricted [37, 39, 12].

  - *Metamorphic Contracts (CREATE2)*: Each redeployment resets storage, and maintaining prior data is difficult, limiting flexibility to make incremental updates to the storage structure [34, 12, 39].
  - *Inherited Storage and Proxy Inherited Storage*: Storage layout follows the inheritance hierarchy, and adding or modifying variables can disrupt the existing structure, reducing flexibility in adjusting stored data.
  - *Proxy-Based Approaches (Basic Proxy, Transparent Proxy, Beacon Proxy, EIP-897, EIP-1967, UUPS)*: The proxy structure requires compatibility across implementation changes, which restricts flexibility in modifying storage as it must follow a consistent layout [47, 35, 69, 13].

- **Moderate Storage Flexibility:**
  - *Strategy Pattern*: While strategy contracts can change logic, the primary contract's storage structure remains largely static, offering limited flexibility for evolving storage needs [15, 38, 71].
  - *EIP-1538*: Functions can be adjusted dynamically, but adding or modifying state variables must be managed carefully to align with existing storage, offering controlled flexibility [59, 60, 12, 51, 14].

- **High Storage Flexibility:**
  - *Eternal Storage*: With storage fully separated from logic, state variables can be added or modified independently, allowing flexible updates to stored data without impacting contract functionality [63, 49, 13].
  - *Proxy Eternal Storage*: Storage is managed separately, which allows independent updates to state variables, providing high flexibility for adjusting storage as needed [34, 70, 64].

- **Very High Storage Flexibility:**
  - *Diamond Pattern (EIP-2535)*: The layout structure prevents conflicts, allowing extensive modifications to stored data without risking disruption to other facets, enabling adaptable storage updates [11, 55, 59].

- *Unstructured Storage*: Unique storage slots for each variable allow new data to be added or modified without affecting existing storage, providing adaptability in adjusting the storage structure.
- *Proxy Unstructured Storage*: With distinct slots for each variable, adding or modifying storage data is straightforward, achieving high adaptability without conflicts [70, 64].

7.2.3. Summary of Flexibility Rankings

As illustrated in Figure 7, the heatmap provides an integrated view of flexibility levels across a range of smart contract upgrade mechanisms. On the X-axis, we depict the two primary dimensions of flexibility (Logic Flexibility and Storage Flexibility), while the Y-axis lists each smart contract upgrade approach. A color gradient represents low to very high flexibility levels, where darker tones highlight higher flexibility and lighter tones signify lower flexibility. We assigned an average flexibility score to each approach to systematically classify and interpret the flexibility data, encompassing both Logic and Storage Flexibility dimensions. Scores range from 1 to 4, with 1 indicating low flexibility and 4 representing very high flexibility. Based on these average scores, we categorized each approach's overall flexibility level as follows:

- **Low Flexibility:**
  - Approaches like *Contract Migration* and *Metamorphic Contracts (CREATE2)* exhibit low flexibility. Both require redeployment for any update, making isolated function modifications or incremental logic refinements impractical. Consequently, changes entail deploying entirely new contracts, resetting the state, and limiting adaptability.
- **Moderate Flexibility:**
  - The *Strategy Pattern* and *Inherited Storage* approaches fall under moderate flexibility. While the Strategy Pattern allows isolated changes through swappable strategy contracts, broader updates are constrained. Inherited Storage enables the introduction of new functions through inheritance but restricts modifications to inherited logic.

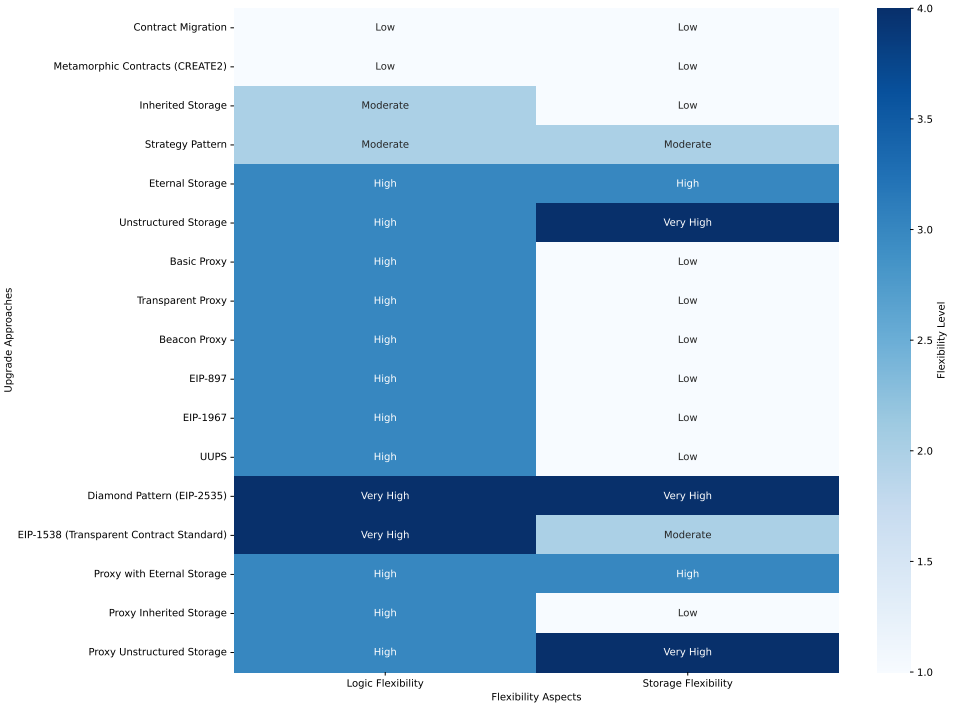

Figure 7: Flexibility levels for smart contract upgrade approaches

- **High Flexibility:**
  - *Proxy-Based Approaches*, including Basic Proxy, Transparent Proxy, Beacon Proxy, EIP-897, EIP-1967, and UUPS, offer high logic flexibility, allowing implementation contract replacements. However, the requirement for consistent storage structures limits their adaptability in storage. *Data Separation Patterns* like Eternal Storage and Unstructured Storage are similarly flexible, as they decouple storage from logic, allowing logic updates without affecting data.
- **Very High Flexibility:**
  - The *Diamond Pattern (EIP-2535)* and *EIP-1538 (Transparent Contract Standard)* approaches are classified as very high flexi-

bility. With extensive modularity and dynamic function management, these approaches permit granular modifications and unrestricted adaptability in both logic and storage.

7.3. Efficiency

Efficiency is a critical factor in smart contract upgradeability, as it directly impacts the cost and performance of contract operations on decentralized systems. According to ISO/IEC 25010, performance efficiency is defined as: *The degree to which a system or component performs its designated functions with minimum consumption of resources.*

Efficiency is analyzed across three key aspects: Approach Deployment Efficiency, Upgrade Deployment Efficiency, and Execution Efficiency. Unlike other themes, efficiency is not classified based on smart contract components, as it depends heavily on the gas consumption of the entire upgrade mechanism rather than individual components. This broader approach allows for a more comprehensive assessment of how various upgrade strategies affect overall resource usage and operational costs.

7.3.1. Approach Deployment Efficiency

Approach Deployment Efficiency refers to the initial setup cost, including the number of base contracts needed to establish each upgrade mechanism. We categorize this efficiency into four levels, where very high indicates a cost-effective setup and low represents a more resource-intensive, costly setup.

- **Very High Approach Deployment Efficiency:**
  - *Contract Migration*: Involves minimal initial setup without proxies, making it cost-effective in deployment [37, 39].
  - *Metamorphic Contracts (CREATE2)*: Simple self-destruction and recreation require minimal setup, keeping initial deployment highly efficient [34, 39, 65].
  - *Strategy Pattern*: Lightweight setup focusing on strategy contracts only, making deployment cost-effective while enabling selective updates[47, 73, 11].
- **High Approach Deployment Efficiency:**

- *Data Separation Approaches (Eternal Storage and Unstructured Storage)*: Setup costs are moderate due to the need for separate storage contracts but remain efficient compared to multi-layered proxies [73, 48, 49, 63].
- *UUPS Proxy*: Includes the upgrade mechanism within the logic contract, resulting in moderate initial setup costs to ensure state and logic are properly aligned [34, 41, 42, 50, 36].

- **Moderate Approach Deployment Efficiency:**
  - *Basic Proxy and Delegate Proxy Approaches (Basic Proxy, EIP-897 Delegate Proxy, EIP-1967 Standard Storage Slots)*: The requirement to deploy both proxy and logic contracts incurs moderate costs but ensures a solid foundation for future upgrades [42, 74, 57, 43].

- **Low Approach Deployment Efficiency:**
  - *Hybrid Approaches (Proxy with Eternal Storage, Proxy Inherited Storage, Proxy Unstructured Storage)*: Dual-proxy configurations and separate storage management add initial complexity and higher gas costs [74, 70, 64].
  - *Beacon Proxy*: Requires multiple contracts, including a beacon and proxy, making it one of the most resource-intensive to set up initially [69, 45].
  - *Diamond Pattern (EIP-2535)*: The multi-facet setup for separate storage management requires extensive initial gas costs, especially when deploying a large number of facets [59, 55].
  - *EIP-1538 (Transparent Contract Standard)*: Complex function mappings and rigorous admin verifications increase setup costs, making it among the least efficient for initial deployment [60, 75].

7.3.2. Upgrade Deployment Efficiency

Upgrade Deployment Efficiency measures the gas costs incurred when deploying a new contract version during upgrades. Similar to Approach Deployment Efficiency, we categorize this into four levels, where very high represents cost-effective upgrades, and low indicates high gas costs and resource use.

- **Very High Upgrade Deployment Efficiency:**
  - *Strategy Pattern*: Deploying only specific strategy contracts results in minimal gas usage for upgrades. This approach allows efficient redeployment without modifying core contracts by isolating the logic within individual strategies [47, 73, 11].
- **High Upgrade Deployment Efficiency:**
  - *Basic Proxy, EIP-897 Delegate Proxy, EIP-1967 Standard Storage Slots*: Only the logic contract needs to be redeployed, and updating the implementation address in the proxy involves minimal costs. Standardized storage slots (EIP-1967) and `delegatecall` used in these proxies add simplicity, avoiding extensive authorization or setup [42, 57, 68].
  - *Data Separation Approaches (Eternal Storage and Unstructured Storage)*: Deployment costs are low, as these approaches only redeploy the logic contract and rely on separated storage [73, 48, 49].
- **Moderate Upgrade Deployment Efficiency:**
  - *UUPS Proxy*: The upgrade functionality resides within the logic contract, requiring a complete redeployment of the logic contract each time, increasing gas consumption compared to basic proxies.
  - *Transparent Proxy*: Additional gas costs are incurred due to admin checks and the need to distinguish between admin and user roles, adding to the complexity of upgrades [41, 42, 50].
  - *Diamond Pattern (EIP-2535)*: While facets allow selective redeployment, managing multiple facets and ensuring consistent storage layout introduces moderate upgrade costs [59, 55].
  - *EIP-1538 (Transparent Contract Standard)*: This approach requires extensive function mappings and admin verifications, resulting in higher upgrade costs [60, 75].
  - *Hybrid Approaches (Proxy with Eternal Storage, Proxy Inherited Storage, Proxy Unstructured Storage)*: Dual-proxy configurations add some complexity but remain manageable due to separated data, keeping costs moderate [74, 70, 64].

- **Low Upgrade Deployment Efficiency:**
  - *Contract Migration and Metamorphic Contracts (CREATE2)*: Redeploying a new version involves significant resources due to manual state migration requirements. Contract migration necessitates data migration and user notification, while Metamorphic Contracts require self-destruction and recreation, leading to additional state management costs [34, 12, 65, 37, 39].

7.3.3. Execution Efficiency

Execution Efficiency evaluates the transaction cost of interacting with or executing the logic contract after an upgrade. Approaches using `delegatecall` typically result in lower gas costs compared to `call`-based methods. We classify execution efficiency into four levels, where very high indicates highly efficient interactions with minimal transaction costs, and low reflects higher gas costs and less efficient execution.

- **Very High Execution Efficiency:**
  - *Contract Migration*: As this approach interacts directly without `delegatecall` or additional proxy layers, it minimizes gas usage during execution, making it highly efficient [37, 76, 39].
  - *Metamorphic Contracts*: These contracts benefit from direct interactions post-deployment without extra execution layers, achieving very low gas usage [34, 12, 65].
- **High Execution Efficiency:**
  - *Basic Proxy and Delegate Proxy Approaches (Basic Proxy, UUPS Proxy, EIP-897 Delegate Proxy, EIP-1967 Standard Storage Slots)*: Execution relies on `delegatecall`, which maintains high efficiency for contract interactions while preserving state [57, 39, 43, 68].
  - *Strategy Pattern*: Direct strategy calls offer high efficiency in execution; however, accessing past state through external calls can slightly increase gas costs but remain within acceptable levels [47, 73, 11].
- **Moderate Execution Efficiency:**

- *Data Separation Approaches (Eternal Storage and Unstructured Storage)*: Execution relies on `call`-based access to storage contracts, adding moderate gas usage due to indirect data access and increased interactions [48, 49, 12, 77, 63].
- *Beacon Proxy*: Gas costs increase due to additional interactions with the beacon during execution, especially when coordinating multiple proxies [57, 39, 43, 68].
- *Transparent Proxy*: Selector checks to distinguish between admin and user calls add notable gas usage, keeping it less efficient than simpler proxy options [41, 42, 50].

- **Low Execution Efficiency:**
  - *Diamond Pattern (EIP-2535)*: Execution involves delegating to multiple facets via a dispatcher, adding significant gas overhead, especially in systems with complex storage needs [59, 55].
  - *Hybrid Approaches (Proxy with Eternal Storage, Proxy Inherited Storage, Proxy Unstructured Storage)*: Dual-proxy structures increase gas usage for execution due to storage separation and added complexity [74, 70, 64].
  - *EIP-1538 (Transparent Contract Standard)*: Each call requires admin verification, resulting in high storage access costs and low efficiency for frequent interactions [60, 75].

7.3.4. Summary of Efficiency Rankings

As illustrated in Figure 8, the heatmap integrates efficiency levels across a spectrum of smart contract upgrade mechanisms. The X-axis represents the three efficiency dimensions: Upgrade Deployment Efficiency, Execution Efficiency, and Approach Deployment Efficiency, while the Y-axis lists each smart contract upgrade approach. A color gradient from light to dark tones indicates efficiency levels, where darker shades represent higher efficiency and lighter shades signify lower efficiency. Using the average efficiency score for each approach, we categorized their overall efficiency as follows:

- **Low Efficiency:**

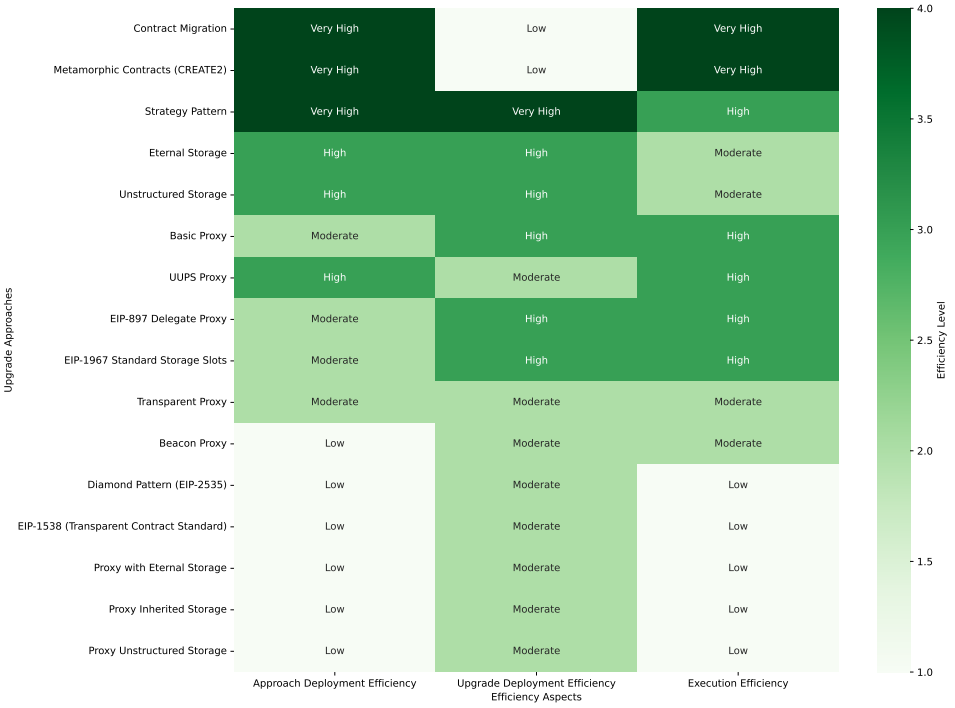

Figure 8: Efficiency levels for smart contract upgrade approaches

  - Approaches like *Diamond Pattern (EIP-2535)*, *EIP-1538 (Transparent Contract Standard)*, *Proxy with Eternal Storage*, *Proxy Inherited Storage*, and *Proxy Unstructured Storage* exhibit low efficiency. These methods introduce complex configurations and high gas costs, especially during upgrades and initial deployments.

- **Moderate Efficiency:**

  - *Beacon Proxy* and *Transparent Proxy* are classified under moderate efficiency. While these approaches provide some flexibility in upgrade deployment, they incur additional gas costs during execution due to checks for admin and user distinctions or beacon interactions.

- **High Efficiency:**

- Approaches including *Contract Migration*, *Metamorphic Contracts (CREATE2)*, *Basic Proxy*, *UUPS Proxy*, *EIP-897 Delegate Proxy*, *EIP-1967 Standard Storage Slots*, *Eternal Storage*, and *Unstructured Storage* are classified as highly efficient. These methods balance deployment and execution costs with straightforward upgrade paths and efficient interaction mechanisms.

- **Very High Efficiency:**

  - The *Strategy Pattern* achieves very high efficiency by deploying only specific strategy contracts for updates, minimizing both deployment and execution costs. Its lightweight setup and direct strategy calls reduce gas consumption, making it an optimal choice for systems needing isolated, targeted updates without redeploying core components.

7.4. Security

Security is a fundamental aspect of smart contract upgradeability, ensuring that only authorized changes occur and the system remains resilient against attacks. According to ISO/IEC 25010, security is defined as: *The degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization.*

We examine security vulnerabilities in smart contract upgradeability across three categories: Logic Vulnerabilities, Storage Vulnerabilities, and Execution Flow Vulnerabilities.

7.4.1. Logic Vulnerabilities

Logic vulnerabilities are critical risks that arise from handling contract logic upgrades or delegate operations.

- **Delegatecall Misuse**

  - *Affected Approaches*: Proxy-Based Approaches, including Basic Proxy, UUPS Proxy, Beacon Proxy, and EIP-897, as well as *Diamond Pattern (EIP-2535)* due to `delegatecall` reliance in facets [34, 48, 74, 43].

- *Partial Mitigation*: The Transparent Proxy Pattern reduces this risk by segregating administrative functions from user functions. However, if the admin account is compromised, this mitigation is less effective [34, 67, 36].

- **Function Selector Clashes**
  - *Affected Approaches*: Proxy-Based Approaches, especially Basic Proxy, Beacon Proxy, and EIP-897, are susceptible to selector clashes. *Diamond Pattern* increases clash risks due to dynamic function handling [54, 67, 69].
  - *Partial Mitigation*: UUPS Proxy reduces the risk by embedding the upgrade function within the logic contract [13, 78]. The Transparent Proxy also minimizes this risk by separating administrative functions from user functions [45, 42].

- **Upgrade Initialization Risks and Bricking**
  - *Affected Approaches*: UUPS Proxy is particularly vulnerable, as it requires upgrade functionality within the logic contract [39, 34, 41, 14]. Metamorphic Contracts (CREATE2) also face bricking risks if redeployed incorrectly [39, 65]. The Diamond Pattern (EIP-2535) is similarly affected, as its upgrade function resides in the logic rather than in the proxy, increasing the risk of improper upgrades [13, 59].

7.4.2. Storage Vulnerabilities

Storage vulnerabilities, such as storage collisions, layout changes across versions, and unauthorized access, affect data integrity.

- **Storage Collisions**
  - *Affected Approaches*: Proxy-Based Approaches like Basic Proxy, Beacon Proxy, and EIP-897, as well as *Diamond Pattern* due to shared storage among facets [50, 34, 54, 48, 52, 42].
  - *Partial Mitigation*: Unstructured Storage or Data Separation patterns mitigate this risk by separating storage from the logic contract [13, 57, 74].

- **Storage Layout Changes Between Versions**

- *Affected Approaches*: All Proxy-Based Approaches face this risk.
- *Mitigation*: Eternal Storage or Unstructured Storage Patterns in Data Separation effectively mitigate this risk by keeping storage in a separate contract.

- **Unauthorized Access to Storage**
  - *Affected Approaches*: Data Separation approaches without strict access control, Hybrid Approaches without adequate restrictions [38, 48, 49, 63].
  - *Mitigation*: Implementing strict access controls and using role-based access control (RBAC) frameworks.

7.4.3. Execution Flow Vulnerabilities

Execution flow vulnerabilities relate to risks in function call control and upgrade mechanisms.

- **Unauthorized Upgrades**
  - *Affected Approaches*: Beacon Proxy, Proxy Patterns with centralized admin control, UUPS Proxy, and Metamorphic Contracts (CREATE2) [38, 48, 49, 63, 37, 43, 79, 58, 67, 36].

- **Centralization Risks**
  - *Affected Approaches*: Beacon Proxy, Proxy with Centralized Admin [34, 67, 79, 62, 11].

7.4.4. Summary of Security Implications by Aspect

Each approach's security risks depend on the specific vulnerabilities addressed in Logic, Storage, and Execution Flow, as shown in Table 5. Below is a ranking for each category based on their security implications:

- **Logic Vulnerabilities Ranking**
  - *Low Vulnerability*: Contract Migration, Strategy Pattern
  - *Moderate Vulnerability*: Transparent Proxy, EIP-1967, Data Separation
  - *High Vulnerability*: Basic Proxy, UUPS Proxy, Diamond Pattern

Table 5: Summary of Security Vulnerabilities in Smart Contract Upgradeability Approaches

Security Category	Vulnerability	Affected Approaches	Partial Mitigations	Severity Ranking
Logic Vulnerabilities	Delegatecall Misuse	Proxy-Based Approaches, Diamond Pattern	Transparent Proxy	High (if delegatecall mishandled)
	Function Selector Clashes	Basic Proxy, Beacon Proxy, EIP-897, Diamond Pattern, EIP-1538	Transparent Proxy, UUPS Proxy	Moderate to High
	Upgrade Initialization Risks and Bricking	UUPS Proxy, Metamorphic Contracts	Transparent Proxy, EIP-1967	Moderate to High
Storage Vulnerabilities	Storage Collisions	Basic Proxy, Beacon Proxy, EIP-897, Diamond Pattern, Inherited Storage	Unstructured Storage, Data Separation, EIP-1967	Moderate to High
	Layout Changes	Proxy-Based Approaches	Hybrid Approaches	Moderate
	Unauthorized Storage Access	Data Separation types, Hybrid Approaches, Metamorphic Contracts	-	High if poorly managed
Execution Flow	Unauthorized Upgrades	Beacon Proxy, Transparent Proxy, UUPS, Metamorphic Contracts	-	Moderate to High
	Centralization Risks	Beacon Proxy, Proxy with Centralized Admin	-	Moderate

- **Storage Vulnerabilities Ranking**
  - *Low Vulnerability*: Unstructured Storage, Data Separation
  - *Moderate Vulnerability*: Transparent Proxy, EIP-1967, Strategy Pattern
  - *High Vulnerability*: Basic Proxy, Contract Migration, Diamond Pattern
- **Execution Flow Vulnerabilities Ranking**
  - *Low Vulnerability*: Contract Migration, Strategy Pattern
  - *Moderate Vulnerability*: Transparent Proxy, EIP-1967
  - *High Vulnerability*: Diamond Pattern, Beacon Proxy, Metamorphic Contracts (CREATE2)

Note: The security of each approach depends heavily on proper implementation, rigorous testing, and adherence to best practices. Even approaches with higher inherent risks can be made secure through diligent development and comprehensive security measures.

7.5. Usability

Usability is essential in designing smart contract upgrade mechanisms, as it directly impacts user experience and trust in the system. According to ISO/IEC 25010, usability is defined as:

The degree to which specified users can use a system to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use.

In smart contracts, usability is evaluated based on two key aspects: Interaction Simplicity and Transparency. These aspects ensure that users can easily interact with the system and clearly understand its operations.

7.5.1. Interaction Simplicity of Smart Contract Upgrade Approaches

Interaction Simplicity refers to how easily users can continue to interact with the contract after an upgrade, particularly regarding whether the contract address stays consistent or needs updating. We rank this aspect into three levels, where low indicates significant challenges for users, and high represents seamless interactions and minimal disruptions.

- **Low Interaction Simplicity:**
  - *Contract Migration*: Each upgrade involves a new contract address, requiring users to switch addresses manually and migrate their data or assets. This complexity in maintaining interaction with the current contract significantly reduces usability [37, 76, 45].
  - *Data Separation Approaches (Inherited Storage, Eternal Storage, Unstructured Storage)*: Similar to Contract Migration, these approaches often require separate data and logic contracts. Users must interact with new addresses post-upgrade, making interaction cumbersome and less intuitive [38, 73, 49, 77].
- **Moderate Interaction Simplicity:**
  - *Metamorphic Contracts (CREATE2)*: While the address remains stable, users may encounter downtime and potential state resets during redeployment. Though interactions are consistent, users may experience disruptions affecting interaction simplicity [34, 12, 45].
  - *Strategy Pattern*: The main contract remains stable, simplifying interactions. However, changes to underlying strategy contracts can modify specific functionalities, requiring slight user adaptation to potential changes in contract behavior [45, 39, 46, 12].

- **High Interaction Simplicity:**
  - *Proxy Approaches*: The proxy maintains a stable address, allowing users to interact consistently without changing their methods. Users experience seamless upgrades as the interface remains unchanged [61, 57, 43, 65].
  - *Hybrid Approaches (Proxy with Eternal Storage, Proxy Inherited Storage, Proxy Unstructured Storage)*: These approaches use proxy mechanisms, preserving a constant address, which ensures continuity and high interaction simplicity [45, 57, 15, 70].

7.5.2. Transparency of Smart Contract Upgrade Approaches

Transparency refers to the extent to which users are aware of upgrades and understand their impact on contract functionality. Similar to Interaction Simplicity, we rank transparency into three levels, where low indicates limited user awareness and understanding, and high signifies full transparency with clear communication and understanding of the upgrade's effects.

- **Low Transparency:**
  - *Contract Migration*: Users must be informed of address changes manually, with a high likelihood of continuing interactions with outdated contracts if not fully notified [37, 76, 45].
  - *Data Separation Approaches (Inherited Storage, Eternal Storage, Unstructured Storage)*: These approaches lack inherent notification mechanisms. Without a registry to point to the latest contract, users are often unaware of upgrades [49, 12].
- **Moderate Transparency:**
  - *Proxy Approaches (Basic Proxy, UUPS, EIP-897 Delegate Proxy, Transparent Proxy)*: Users can detect upgrades by reviewing proxy transactions, but these require active checking. Users may not be directly informed of updates without prior knowledge of where to look [61, 67, 68, 65, 79].
  - *Hybrid Approaches (Proxy with Eternal Storage, Proxy Inherited Storage, Proxy Unstructured Storage)*: Similar to basic proxies, users can determine upgrades through transaction records, though direct awareness depends on proactive checking [45, 57, 15].

- **High Transparency:**
  - *EIP-1967 Standard Storage Slots, Diamond Pattern (EIP-2535), EIP-1538 (Transparent Contract Standard), Beacon Proxy*: These approaches emit events during upgrades, notifying users of changes and the new implementation address. This proactive communication enhances user awareness of modifications and preserves trust through transparent upgrade mechanisms [80, 60, 59, 55, 14].

7.5.3. Summary of Usability Rankings

Figure 9 illustrates that the heatmap integrates usability levels across a spectrum of smart contract upgrade mechanisms. The X-axis represents the two primary usability dimensions—Interaction Simplicity and Transparency—while the Y-axis lists each smart contract upgrade approach. A color gradient from light to dark tones indicates usability levels, where darker shades represent higher usability and lighter shades signify lower usability. Using the average usability score for each approach, we categorized their overall usability as follows:

- **Low Usability:**
  - Approaches like *Contract Migration*, *Inherited Storage*, *Eternal Storage*, and *Unstructured Storage* exhibit low usability. These methods introduce interaction complexity as they require users to interact with different addresses post-upgrade and lack clear mechanisms for notifying users of upgrades.
- **Moderate Usability:**
  - The *Metamorphic Contracts (CREATE2)* and *Strategy Pattern* approaches fall under moderate usability. While Metamorphic Contracts maintain a stable address, they may involve downtime and potential state resets during redeployment. The Strategy Pattern retains a stable main contract address, simplifying interaction, though underlying strategy changes may require slight user adaptation.
- **High Usability:**

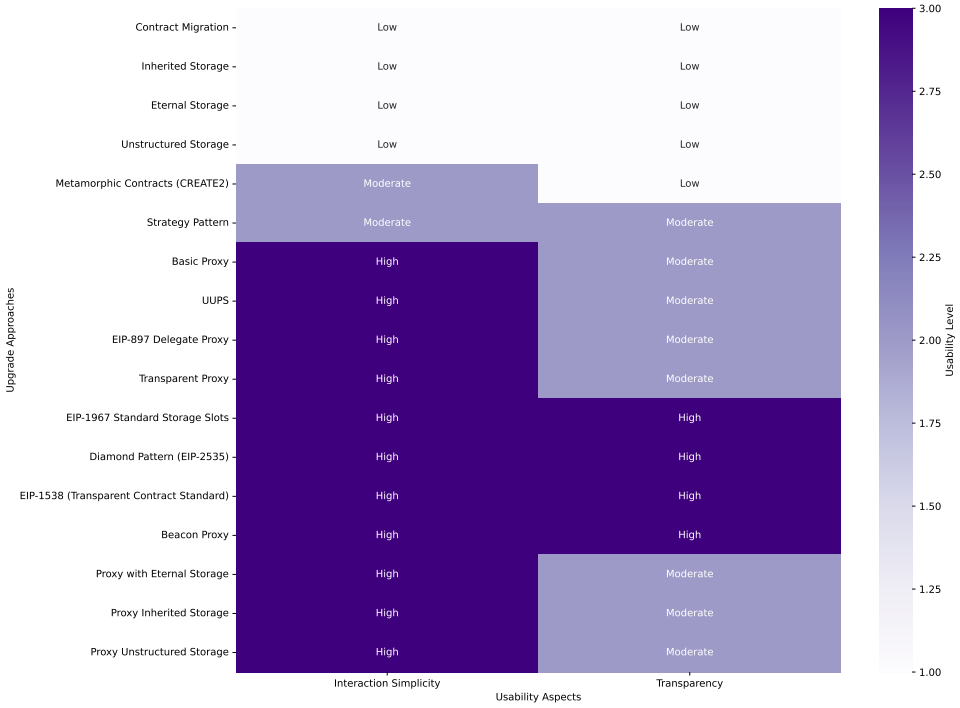

Figure 9: Usability levels for smart contract upgrade approaches

– A variety of approaches, including *Basic Proxy*, *UUPS*, *EIP-897 Delegate Proxy*, *Transparent Proxy*, *EIP-1967 Standard Storage Slots*, *Diamond Pattern (EIP-2535)*, *EIP-1538 (Transparent Contract Standard)*, *Beacon Proxy*, *Proxy with Eternal Storage*, *Proxy Inherited Storage*, and *Proxy Unstructured Storage* are classified as highly usable. These approaches offer stable addresses, enabling users to interact seamlessly post-upgrade and benefit from moderate to high transparency.

7.6. RQ3 Answer

To address RQ3: What are the benefits and limitations of each upgrading approach?, we evaluated each approach based on attributes such as complexity, flexibility, efficiency, security, and usability. Figure 10 presents a heatmap that offers a comprehensive overview of the performance of different smart contract upgrade approaches across these themes. Choosing the

Figure 10: Summary of smart contract upgrade approaches accross all themes

right approach often means balancing these themes to fit a project's unique needs. For projects that prioritize simplicity and safety, Strategy Pattern and Unstructured Storage are strong options. The Strategy Pattern stands out for its low complexity and minimal security risks, making it ideal for applications where straightforward updates are needed. Unstructured Storage, though more complex, offers high flexibility and good efficiency while keeping security risks low. However, it has lower usability due to its complex implementation. For projects looking for a balance across most themes, UUPS Proxy and Transparent Proxy offer reliable solutions. UUPS Proxy combines low complexity with high efficiency, providing moderate flexibility and usability, though it needs careful handling to address some security risks. Transparent Proxy has moderate complexity and security risks but delivers strong efficiency and usability, making it a practical and widely used choice. Both proxies keep the contract address the same during upgrades, so users

do not need to update their interactions, which adds to usability.

For projects needing high flexibility, especially with large and complex contracts, Diamond Pattern excels by allowing modular upgrades. However, this comes with higher complexity and security risks, requiring skilled developers and thorough security practices. Basic Proxy and EIP-1967 Proxy offer a balanced mix of moderate complexity, flexibility, and security, along with good efficiency and usability, making them suitable for various types of projects. Ultimately, selecting the right upgrade approach involves understanding the trade-offs between themes. The heatmap in Figure 10 serves as a helpful visual guide to compare these approaches and find the best fit based on the specific needs of the project.

8. Discussion

This section synthesizes insights on smart contract upgradeability, focusing on governance models, lifecycle management, and the decision "to upgrade or not." While upgrades offer adaptability, they do not inherently guarantee improved security or usability. Each theme is supported by case studies and best practices, highlighting the complexities and trade-offs of smart contract upgradeability.

8.1. Governance Models in Upgradeability

Smart contract governance utilizes on-chain rules to establish decision-making processes and manage upgrades with the goals of adaptability, security, and transparency. Governance structures significantly influence user trust, as they determine who controls the upgrade process and how transparently changes are implemented. However, governance in smart contracts faces critical challenges, particularly in balancing decentralization with effective oversight and quick response to urgent issues.

8.1.1. Centralized Governance Models

Centralized models, commonly used in patterns like the Transparent Proxy and Beacon Proxy, consolidate upgrade control within a single administrator or a small group, allowing for rapid decision-making. For instance, OpenZeppelin centralizes control under an admin account for efficient upgrades.[3] While efficient, centralized governance can create vulnerabilities, as

[3] `https://docs.openzeppelin.com/contracts/4.x/api/proxy`

shown by the PAID Network hack. In this incident, attackers compromised the admin's private key, resulting in unauthorized token minting and significant user losses. This underscores the inherent risks of centralization, as a single point of failure can lead to catastrophic security breaches and erode user trust.[4]

8.1.2. Decentralized Governance Models

Decentralized models often rely on community voting or multi-signature controls, where multiple stakeholders hold decision-making power to reduce reliance on any one entity. Compound and MakerDAO, two decentralized finance (DeFi) protocols, utilize token-based voting to empower their communities in governance decisions.[5][6] However, decentralized governance faces challenges like low voter participation and delayed responses. For instance, a critical bug in Compound once mistakenly distributed $90 million in COMP tokens due to the delay needed to reach consensus on corrective action, revealing the limitations of decentralized models in urgent situations.[7] To mitigate this, Compound employs timelocks to give users time to review and object to upgrades, although this setup sacrifices responsiveness.

8.1.3. Reliance on Oracles and External Data

Some smart contract systems depend on oracles for off-chain data, such as asset prices in DeFi protocols. Oracles introduce vulnerabilities as they rely on external data sources that can be manipulated, compromising the contract's security and decentralization. For example, inaccurate or malicious data from oracles has led to significant losses in DeFi protocols, emphasizing the need for robust governance and verification mechanisms to manage these dependencies securely.[8]

8.1.4. Distinct Governance Models in Platforms

Blockchain platforms adopt a variety of governance models to manage upgrades, balancing flexibility, decentralization, and transparency. For in-

[4] https://blog.paidnetwork.com/

[5] https://compound.finance/

[6] https://makerdao.com/en/governance

[7] https://www.icba.org/newsroom/blogs/main-street-matters/2021/11/12/the-challenges-and-risks-of-smart-contracts

[8] https://corpgov.law.harvard.edu/2020/04/25/an-introduction-to-smart-contracts-and-their-potential-and-inherent-limitations

stance, Tezos[9] enables upgrades through community voting without replacing the entire contract, promoting flexibility while maintaining decentralization. Similarly, Polkadot[10] employs an on-chain governance system where token holders and a council collaborate to propose and approve upgrades. In contrast, EOS[11] centralizes control by allowing select account groups to manage upgrades, raising transparency and decentralization concerns. These examples illustrate the diversity in governance structures, each with trade-offs in security, efficiency, and user trust.[12]

8.2. Best Practices in Governance

Implementing governance in smart contracts requires best practices to balance security, transparency, and decentralized control:

- **Define Clear Rules and Stakeholder Responsibilities:** Establishing transparent, well-documented processes for decision-making is crucial. Clearly defining roles, voting protocols, and decision criteria helps ensure a fair governance process, enhancing user trust and minimizing ambiguity.[13]

- **Multi-Signature Wallets and Timelocks:** Multi-signature wallets prevent unilateral control by requiring multiple approvals for sensitive actions, like upgrades. Timelocks introduce delays that allow stakeholders to review and, if necessary, intervene. Protocols like Uniswap and Compound employ multisig setups and timelocks to balance rapid response capabilities with robust oversight.[14]

- **Continuous Auditing and Monitoring:** Regular audits are essential for identifying vulnerabilities in governance mechanisms, especially as protocols evolve. Monitoring transaction activities through alerts

[9] https://tezos.com/

[10] https://polkadot.network/

[11] https://eosnetwork.com/

[12] https://www.dutchblockchaincoalition.org/

[13] https://www.doubloin.com/what-is-smart-contract-governance-make-on-chain-decisions

[14] https://blog.openzeppelin.com/smart-contract-security-guidelines-4-strategies-for-safer-governance-systems

and off-chain event logs enables quick responses to unusual activities, improving security and operational resilience.[15]

- **Stakeholder Engagement:** Voting systems, common in DeFi platforms, allow token holders to participate in governance. Although beneficial, these systems are vulnerable to manipulations like flash loans, where attackers temporarily gain control over voting. Ethereum.org emphasizes the need for safeguards in decentralized voting mechanisms to protect governance integrity.[16]

- **Emergency Stop and Fail-Safe Mechanisms:** An emergency stop allows authorized entities to pause critical operations during security incidents. These functions should ideally be decentralized to prevent misuse, providing a fail-safe in high-risk situations without relying on centralized control.[17]

- **Adapt and Improve Through Feedback:** Regularly reassessing governance based on user feedback and evolving needs helps refine the model. Continuous improvements align governance structures with community expectations and long-term project goals, sustaining trust and adaptability.[18]

8.3. Future Research Directions

Despite the importance of governance in decentralized applications, empirical studies on governance models in smart contracts remain limited. Research is crucial to evaluate the practical implications of governance structures on security, user trust, and project stability. Specific areas for future investigation include:

- **Comparative Studies of Centralized vs. Decentralized Models:** Evaluating the cost-benefit of different governance structures would

[15] `https://www.webisoft.com/blog/best-practices-for-smart-contract-security`

[16] `https://ethereum.org/en/developers/docs/smart-contracts/security`

[17] `https://ethereum.org/en/developers/docs/smart-contracts/security`

[18] `https://www.doubloin.com/what-is-smart-contract-governance-make-on-chain-decisions`

provide insights into optimizing models for security and efficiency. Empirical studies could examine user trust across various structures, assessing the impact of centralized versus decentralized approaches on engagement and security.

- **Adaptive Timelock Mechanisms:** Adaptive timelocks that adjust based on the urgency or scale of the upgrade could help strike a balance between responsiveness and security. Research into such mechanisms, particularly in DeFi, would support flexible governance protocols that align with both community needs and security requirements.

- **Lifecycle Tracking and Real-Time Monitoring Tools:** The development of tools that offer real-time monitoring and lifecycle tracking for governance decisions could empower stakeholders by increasing transparency. These tools would allow the community to actively monitor governance activities, fostering a transparent and secure environment.

- **Stakeholder Engagement and Voting Efficiency:** Studies focused on improving voter participation and engagement, potentially through incentivized voting models, would provide actionable insights into strengthening decentralized governance. Given the challenges of voter apathy, research on optimal incentive structures could increase participation and ensure that community-driven governance remains effective and representative.

- **Best Practices for Oracle Governance:** With the increasing reliance on oracles in smart contract governance, research on best practices for oracle management is needed. Empirical studies examining oracle reliability, security, and how oracle data impacts contract outcomes would contribute to more robust governance structures that minimize external dependencies.

8.4. Lifecycle Management: Ensuring Stability and Reducing Complexity

Lifecycle management addresses how contracts evolve from deployment through updates, requiring strategic planning to maintain stability and user trust. Key approaches include single-time upgradeability, routine upgradeability through proxy patterns, and partial immutability with the option to "brick" contracts. Each has specific implications for usability, security, and complexity.

8.4.1. Single-Time Upgradeability

Single-time upgradeability, seen in methods like Contract Migration and Metamorphic Contracts (CREATE2), involves deploying a new contract each time an upgrade is necessary. This approach avoids complexities associated with `delegatecall` or layered proxy mechanisms. However, it requires users to adopt new contract addresses, potentially creating confusion and increasing the risk of interacting with outdated or malicious contracts.

One relevant example is Synthetix's[19] migration process, in which contracts must be fully redeployed with each upgrade. This approach necessitates detailed communication with users, as failure to update addresses can result in interactions with obsolete contracts. Synthetix's approach underscores the challenge of maintaining user engagement and trust in migration models, where user adaptation costs remain a concern.

8.4.2. Routine Upgradeability through Structured Patterns

Routine upgradeability models allow frequent updates without altering the contract address, enhancing usability by maintaining a stable address interface. Although proxy patterns like UUPS (Universal Upgradeable Proxy Standard) are commonly associated with this model, other approaches, such as the Strategy Pattern or Diamond Standard (EIP-2535), also enable structured upgrades. In Audius,[20] a storage collision in a proxy pattern led to a governance hack, underscoring the importance of rigorous storage layout management in upgradeable contracts. Such incidents demonstrate the need for careful structuring in routine upgrades to prevent data collisions and unauthorized access.

8.4.3. Partial Immutability

Partial immutability, orbricking, allows developers to permanently remove upgradeability from a contract, effectively freezing its state. This approach can increase user trust by ensuring the contract remains stable and unalterable once it reaches maturity. However, bricking can inadvertently prevent necessary upgrades or security patches if misapplied.

Case Study: UUPSUpgradeable Vulnerabilities. A notable instance of accidental loss of upgradeability involved OpenZeppelin's UUPSUpgradeable

[19] https://developer.synthetix.io/contracts/

[20] `https://audius.co/`

contracts. In 2021, a vulnerability in `_authorizeUpgrade` allowed unauthorized upgrades on uninitialized contracts, bypassing security checks. This oversight impacted several projects and exposed assets to risk, which was resolved in OpenZeppelin Contracts v4.3.2 by adding an `onlyProxy` modifier to restrict upgrade calls directly on the implementation contract.[21][22]

Trade-Offs of Bricking. While bricking can ensure stability, it carries the risk of blocking essential maintenance if used prematurely. Balancing the security benefits of immutability with the need for potential updates is essential for effective lifecycle management in smart contracts.

8.4.4. Future Research Directions

As smart contracts evolve, lifecycle tracking and version control tools tailored to blockchain's decentralized, often immutable nature are urgently needed.

- **Lifecycle Tracking and Monitoring Tools:** These tools would offer real-time insights into contract states, including upgrade history, current versions, and any upcoming changes, providing users and developers with transparency across all contract stages. This visibility would allow users to make informed decisions about interacting with the contract, fostering trust and confidence in its long-term stability.
- **Version Control and Standardization Protocols:** Standardized protocols are essential to streamline upgrade management and support rollback capabilities, much like traditional software systems. These would enable developers to track upgrades, review prior versions, and maintain continuity across complex, long-term projects, improving security and usability.
- **Empirical Research on Partial Immutability and Lifecycle Strategies:** Empirical studies exploring the timing, effectiveness, and user perception of bricking can provide insight into how and when contracts should transition to a fixed state. This research could guide user adaptation across different lifecycle strategies, including the impact of

[21] `https://forum.openzeppelin.com/t/uupsupgradeable-vulnerability-post-mortem/15680`

[22] `https://iosiro.com/blog/uups-proxy-security-review`

bricking on user confidence and contract sustainability. Case studies of both successful and failed bricking implementations would offer practical examples, aiding developers in understanding the best timing and conditions for applying partial immutability.

8.5. To Upgrade or Not: The Impact on User Trust and Security

The decision to make a smart contract upgradeable is crucial, balancing the advantages of flexibility with the foundational blockchain principle of immutability. While upgradeability enables developers to address vulnerabilities, introduce new functionalities, and ensure compatibility with evolving standards, it also introduces risks that challenge traditional assumptions of security and trust within the blockchain ecosystem. This section explores the benefits and risks associated with upgradeable contracts, along with best practices and directions for future research.

8.5.1. Benefits of Upgradeability

Bug Fixes and Security Improvements. The ability to address bugs and vulnerabilities post-deployment is a major advantage of upgradeable contracts. In decentralized finance (DeFi), where considerable value is often locked in contracts, vulnerabilities can lead to severe financial risk. Selective upgradeability, managed carefully, supports both security and efficiency by addressing security flaws without extensive disruptions to the contract's core functionality.[23]

Adaptability to New Features and Scalability. Upgradeability facilitates the addition of new functionalities and supports scalability, particularly for long-term projects. Patterns such as the Diamond Pattern employ modular facets, which allow developers to expand or modify contract functions without disrupting the entire codebase. This modularity reduces costs and supports growth, as developers can gradually scale their projects without needing full replacements, making it ideal for applications with evolving requirements.[24]

Compatibility with Evolving Standards. Long-term projects benefit from upgradable contracts to ensure compliance with industry standards or regulatory

[23] `https://ethereum.org/en/developers/docs/smart-contracts/upgrading/`
[24] `https://blog.openzeppelin.com/the-state-of-smart-contract-upgrades/`

shifts. This flexibility allows adjustments that maintain compatibility without requiring users to interact with a new contract address, thereby enhancing usability and user retention.[25]

8.5.2. Risks of Upgradeability

Challenges to Immutability and User Trust. The ability to modify contracts after deployment directly challenges the immutability that users expect in blockchain systems. Users often rely on the assurance that a contract will perform exactly as initially deployed. However, with upgradeability, users must trust the developers or administrators managing the upgrade functions. This added layer of trust can compromise the perceived neutrality and trustworthiness of the contract, as users may fear unauthorized or poorly managed changes.[26]

Security Vulnerabilities in Delegatecall and Storage Collisions. Upgradeable contracts that rely on `delegatecall` (proxy-based approaches) inherit certain risks, including storage collisions and unauthorized access. A well-known example is the Audius hack, where a storage collision within a proxy contract allowed attackers to manipulate contract settings and access treasury funds. This incident underscores the need for precise storage layout management to avoid unauthorized actions, which can jeopardize contract security and disrupt user trust.[27]

Irreversible Bricking Through Partial Immutability. Partial immutability, or "bricking," is a mechanism by which developers remove upgradeability from a contract, rendering its state permanent. This approach can build user trust by guaranteeing that a contract will become immutable after achieving stability. However, improper or premature use of bricking can lock out essential functions or block security updates, presenting a significant risk.[28]

8.5.3. Best Practices for Mitigating Risks in Upgradeable Contracts

To manage the security and trust challenges that accompany upgradeable contracts, projects adopt a series of best practices:

[25] `https://archive.trufflesuite.com/post/upgrading-smart-contracts-should-you-do-it-and-how`

[26] `https://hacken.io/research/upgrading-smart-contracts-explanation-security-concerns`

[27] `https://audius.co/`

[28] `https://ethereum.org/en/developers/docs/smart-contracts/upgradeability`

- **Decentralized Governance for Security and Transparency:** Implementing decentralized governance models, such as multi-signature wallets and token-based voting, helps distribute control over upgrades. This minimizes the risks associated with a single point of failure, as decisions are collectively managed, ensuring no single party can unilaterally control upgrades.

- **Careful Use of Event Logging for Transparency:** Emitting on-chain events whenever an upgrade is proposed or completed improves visibility, allowing users to monitor contract changes in real time. These events enhance transparency and accountability by providing a clear audit trail for all upgrades, which reassures users and improves contract trustworthiness.[29]

- **Use of Timelocks and Adaptive Control Mechanisms:** Timelocks provides a delay between when an upgrade is proposed and when it takes effect, giving users time to review and, if necessary, exit the contract. Adaptive timelocks, which adjust based on the severity or type of upgrade, can optimize the balance between security and responsiveness, especially in high-value contracts where time for community review is critical.[30]

8.5.4. Future Research Directions

Despite the prevalent use of upgradeable smart contracts, limited empirical research addresses the impact of upgradeability on user trust, security, and long-term project sustainability. Given the growing importance of these mechanisms in DeFi and blockchain applications, future studies could yield critical insights into best practices and improvement areas:

- **Empirical Studies on User Trust in Upgradeable Contracts:** To understand how users perceive and interact with upgradeable versus immutable contracts, empirical research should explore user attitudes toward trust and control in different upgradeability models. Surveys and behavioral studies assessing user willingness to engage with up-

[29] `https://metana.io/blog/upgrading-smart-contracts-heres-all-you-need-to-know`

[30] `https://compound.finance/governance`

gradeable contracts can inform developers on the transparency and control mechanisms users find most reassuring.

- **Impact of Upgradeability on Contract Security and Risk Mitigation:** As security incidents continue to highlight vulnerabilities in upgradeable systems, empirical studies focused on specific risks, such as `delegatecall` exploits and storage collisions, are needed. Case studies on security incidents like the Audius and Wormhole hacks can guide the development of risk assessment frameworks, ensuring that upgradeability mechanisms enhance security without compromising functionality.[31]

9. Threats to Validity

While our study aims for thoroughness and accuracy, potential internal threats to validity could impact the reliability of our findings. Addressing these threats is crucial to understanding the limitations of our research and the measures taken to mitigate potential biases. Below, we discuss the main types of validity threats: internal validity, external validity, construct validity, and reliability, highlighting specific concerns and the strategies used to manage them.

Internal validity in our study faces potential threats primarily due to the inclusion of both academic and grey literature. Grey literature, such as technical blogs and community forums, varies in credibility and quality, which could influence our conclusions. Although we employed systematic search strategies and predefined inclusion and exclusion criteria, the subjective nature of selecting grey literature may have introduced bias. We attempted to reduce this risk by conducting quality assessments using established frameworks and involving multiple reviewers in the selection process. Additionally, data extraction and thematic analysis are subject to interpretation, which may introduce bias. To mitigate this, we had multiple reviewers extract data independently and resolve discrepancies through consensus meetings, alongside calculating inter-rater reliability to ensure consistency. However, the inherent subjectivity of qualitative research may still affect the categorization and evaluation of upgrade approaches.

External validity is a concern as our focus on Ethereum smart contracts may limit our findings' generalizability to other blockchain platforms with

[31] `https://wormhole.com/`

different architectures, consensus mechanisms, or programming languages. This specialization could restrict the broader applicability of our conclusions. Moreover, while we conducted extensive searches across various databases and utilized citation tracking to minimize publication bias, the dynamic nature of the field may still influence the representation and emphasis of specific upgrade approaches, potentially affecting the balance of our analysis.

Construct validity in our study could be affected by the varied credibility of grey literature. Despite systematic methods and predefined criteria for inclusion, the subjective nature of selecting grey literature sources may impact how well the study captures the broader landscape of smart contract upgradeability. To address this, we performed quality assessments to maintain a consistent standard, though the variability in grey literature remains a consideration that could influence the quality and depth of our findings.

Reliability, or the consistency of our findings, is affected by the use of grey literature, which may not always be archived or easily accessible in the future. To improve reproducibility, we documented the output of these sources in detail and set clear criteria for selecting and assessing them. This documentation helps future researchers replicate or expand on our work. During our study, we encountered instances where some sources were no longer available online, which led to their exclusion from our analysis. Furthermore, the rapid pace of change in blockchain technology also affects the reliability of our study. New upgrade mechanisms or variations may have emerged after our knowledge cutoff in July 2024, which could influence the relevance of our findings. Regular updates and ongoing research are necessary to ensure our analysis remains up-to-date. Researchers and practitioners should consider the timing of our study when applying its conclusions to future projects.

10. Conclusion

Smart contract upgradeability is critical to blockchain development, addressing the need for flexibility and adaptability in decentralized applications. Through our Multivocal Literature Review, we have systematically classified and analyzed the existing approaches to smart contract upgrades, bridging the gap between academic research and industry practices. Our study identifies two primary categories of upgrade mechanisms: full upgrade approaches, which involve redeploying contracts, and partial upgrade approaches, which modify specific components while preserving the contract state. We have provided a unified terminology to standardize the discussion of these meth-

ods. We highlight the trade-offs inherent in selecting an upgrade mechanism by evaluating each approach against software quality attributes: complexity, flexibility, efficiency, security, and usability. For instance, proxy-based approaches offer moderate flexibility and maintain address consistency but introduce complexity and potential security risks. Data separation methods simplify storage management but may impact interaction simplicity. Our analysis underscores the importance of considering project-specific requirements when selecting an upgrade approach. Developers must balance flexibility and adaptability with the principles of security and user trust inherent in blockchain technology. Furthermore, our discussion of governance models and lifecycle management emphasizes the role of transparent and robust governance in maintaining user trust during contract upgrades. Future research should focus on empirical studies that evaluate the real-world impact of different upgrade mechanisms on security and user trust and the development of tools and standards that facilitate secure and efficient smart contract upgrades. By advancing the understanding of smart contract upgradeability, we aim to contribute to developing more resilient and adaptable blockchain applications.

Appendix A. Collected Data from Initial Grey Literature Search

Details of all grey literature sources identified during the initial search phase, including those reviewed but not selected, are available online via this link.

Appendix B. Collected Data from Initial Academic Literature Search

Details of all academic literature sources identified during the initial search phase, including those reviewed but not included in the final analysis, are available online via this link.

Appendix C. Detailed Inclusion and Exclusion Process

The full list of academic and grey literature sources (both included and excluded), including reviewers' decisions, and reasons for inclusion or exclusion, is available online at this link.

Appendix D. Comprehensive List of Included Sources

List of all sources included in the study, both academic and grey literature, with details such as title, author, and year of publication are available online via this link.

Appendix E. Quality Assessment of Grey Literature Sources

Details about the quality scores for all assessed grey literature sources, including the breakdown of scores for each criterion are available online via this link.

Appendix F. Pilot Study Results for Data Extraction Consistency

Summary of the pilot study results are available online via this link.

Appendix G. Standardized Data Extraction Form Template

The template used for extracting data from sources, outlining columns and definitions for each data point is available online via this link.

Appendix H. Full Extracted Data from Included Sources

Spreadsheet summarizing the extracted data from all included sources, covering their characteristics and identified benefits or limitations are available online via this link.

Appendix I. Data Synthesis for RQ1

Overview of the thematic analysis conducted for RQ1, including initial coding and classification of smart contract upgrade approaches into Full and Partial Upgrade categories , is available online via this link.

Appendix J. Data Synthesis for RQ2

The framework analysis for RQ2 outlines data mapping to core components (address, logic, storage, execution flow), indexing, and thematic categorization. Further resources, including the full framework and analysis process, are accessible online via this link.

Appendix K. Data Synthesis for RQ3

The thematic analysis steps for RQ3 include independent coding of benefits and limitations, card-sorting for theme grouping, and alignment with ISO/IEC 25010 quality criteria. Detailed steps of the analysis is available online via this link.

Appendix L. Mapping of Standardized Terms to Literature

A comprehensive table mapping various terms in the literature to the standardized terminology used in this study for consistency and cross-referencing is available online via this link.

References

[1] M. M. Lehman, "Programs, life cycles, and the laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[2] P. Sampaio, J. Silva, and R. Almeida, "Safe evolution of smart contracts," in *Proceedings of the International Conference on Blockchain Technology*, 2023.

[3] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, and M. Imran, "An overview on smart contracts: Challenges, advances and platforms," *Future Generation Computer Systems*, vol. 105, pp. 475–491, 2020.

[4] V. Buterin, "A next-generation smart contract and decentralized application platform." `https://ethereum.org/en/whitepaper/`, 2014.

[5] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2018.

[6] A. Antonino, D. Sorrentino, and M. Rossi, "Specification is law: Safe creation and upgrade of ethereum smart contracts," *arXiv preprint arXiv:2205.07529*, 2022.

[7] D. Siegel, "The dao: A new era of crowdfunding." `https://hbr.org/2016/06/the-dao-a-new-era-of-crowdfunding`, 2016.

[8] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *International Conference on Security and Privacy in Communication Networks*, 2018.

[9] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts," in *International Conference on Principles of Security and Trust (POST)*, 2017.

[10] Y. Chen, Z. Zheng, E. C. H. Ngai, Q. Zheng, and Y. Zhou, "Soda: A generic online detection framework for smart contracts," in *Proceedings of the Network and Distributed System Security Symposium*, 2020.

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

[12] M. Salehi, J. Clark, and M. Mannan, "Not so immutable: Upgradeability of smart contracts on ethereum," in *International Conference on Financial Cryptography and Data Security*, pp. 539–554, Springer, 2022.

[13] W. E. Bodell III, S. Meisami, and Y. Duan, "Proxy hunting: understanding and characterizing proxy-based upgradeable smart contracts in blockchains," in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 1829–1846, 2023.

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

[15] V. C. Bui, S. Wen, J. Yu, X. Xia, M. S. Haghighi, and Y. Xiang, "Evaluating upgradable smart contract," in *2021 IEEE International Conference on Blockchain (Blockchain)*, pp. 252–256, IEEE, 2021.

[16] J. Cano-Benito, A. Cimmino, and R. García-Castro, "Toward the ontological modeling of smart contracts: A solidity use case," *IEEE Access*, vol. 9, pp. 140156–140172, 2021.

[17] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, vol. 3, no. 2, pp. 77–101, 2006.

[18] J. Ritchie and L. Spencer, "Qualitative data analysis for applied policy research," in *Analyzing Qualitative Data*, pp. 173–194, Routledge, 1994.

[19] A. Blanco, Z. Shi, D. Roy, and Z. Zhao, "Improving the resiliency of decentralized crowdsourced blockchain oracles," *Journal of Blockchain Technology*, pp. 3–17, 2023.

[20] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts," *IEEE Symposium on Security and Privacy*, 2020.

[21] G. Gürsoy, C. Brannon, and M. Gerstein, "Using ethereum blockchain to store and query pharmacogenomics data via smart contracts," *BMC Medical Genomics*, vol. 13, no. 1, 2020.

[22] A. Vacca, "Functional suitability assessment of smart contracts: A survey and first proposal," *Journal of Software Evolution and Process*, vol. 36, no. 7, 2023.

[23] I. Reshi, M. Khan, S. Shafi, S. Sholla, A. Assad, and H. Shafi, "Ai-powered smart contracts: The dawn of web 4.0," *TechRxiv Preprint*, 2023.

[24] J. Godoy, J. Galeotti, D. Garbervetsky, and S. Uchitel, "Predicate abstractions for smart contract validation," *ACM Transactions on Software Engineering and Methodology*, 2022.

[25] S. Khan, F. Loukil, C. Ghedira-Guegan, E. Benkhelifa, and A. Bani-Hani, "Blockchain smart contracts: Applications, challenges, and future trends," *Peer-to-Peer Networking and Applications*, vol. 14, no. 5, pp. 2901–2925, 2021.

[26] M. Soud, S. Helgason, G. Hjálmtỳsson, and M. Hamdaqa, "Trustvote: On elections we trust with distributed ledgers and smart contracts," in *2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pp. 176–183, IEEE, 2020.

[27] T. Chen, X. Li, L. Wang, and X. Zhang, "Under-optimized smart contracts devour your money," in *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017.

[28] Y. He, H. Dong, H. Wu, and Q. Duan, "Formal analysis of reentrancy vulnerabilities in smart contract based on cpn," *Electronics*, vol. 12, no. 10, p. 2152, 2023.

[29] G. A. Oliva, A. E. Hassan, and Z. M. Jiang, "An exploratory study of smart contracts in the ethereum blockchain platform," *Empirical Software Engineering*, vol. 25, pp. 1864–1904, 2020.

[30] C. Connors and D. Sarkar, "Review of most popular open-source platforms for developing blockchains," in *2022 fourth international conference on blockchain computing and applications (bcca)*, pp. 20–26, IEEE, 2022.

[31] M. Hamdaqa, L. A. P. Met, and I. Qasse, "icontractml 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms," *Information and Software Technology*, vol. 144, p. 106762, 2022.

[32] V. Garousi, M. Felderer, and M. V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering," *Information and Software Technology*, vol. 106, pp. 101–121, 2019.

[33] K. Krippendorff, *Content Analysis: An Introduction to Its Methodology*. Sage Publications, 2nd ed., 2004.

[34] OpenZeppelin, "The state of smart contract upgrades," 2020. Accessed: 2024-10-04.

[35] Chainlink, "Upgradable smart contracts guide," 2022. Accessed: 2024-10-04.

[36] SCSFG, "Upgradeability," 2023. Accessed: 2024-10-04.

[37] H. Noon, "Smart contract versioning," 2020. Accessed: 2024-10-04.

[38] E. Foundation, "Upgrading smart contracts." `https://ethereum.org/en/developers/docs/smart-contracts/upgrading/`, 2024. Accessed: 2024-11-01.

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

[40] H. Noon, "How to make smart contracts upgradable," 2018. Accessed: 2024-10-04.

[41] L. Blog, "Using the uups proxy pattern to upgrade smart contracts." `https://blog.logrocket.com/using-uups-proxy-pattern-upgrade-smart-contracts/`, 2024. Accessed: 2024-11-01.

[42] QuickNode, "An introduction to upgradeable smart contracts." `https://www.quicknode.com/guides/knowledge-base/an-introduction-to-upgradeable-smart-contracts`, 2024. Accessed: 2024-11-01.

[43] J. C. Lopez Marin, *Upgradeable Smart Contracts design patterns for Dapps Architectures*. PhD thesis, Dublin, National College of Ireland, 2022.

[44] M. Salehi, *An analysis of upgradeability, oracles, and stablecoins in the ethereum blockchain*. PhD thesis, Concordia University, 2022.

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

[46] OpenZeppelin, "Security in upgrades of smart contracts." `https://www.youtube.com/watch?v=5WE6PEc305w`, 2021. Accessed: 2024-10-04.

[47] Coinmonks, "Proxy pattern and upgradeable smart contracts." `https://medium.com/coinmonks/proxy-pattern-and-upgradeable-smart-contracts-45d68d6f15da`, 2024. Accessed: 2024-11-01.

[48] T. of Bits, "Contract upgrade anti-patterns," 2018. Accessed: 2024-10-04.

[49] Indorse, "Ethereum upgradeable smart contract strategies," 2018. Accessed: 2024-10-04.

[50] Moralis, "What are upgradable smart contracts? full guide," 2022. Accessed: 2024-10-04.

[51] T. Wiesner, "Smart contract upgrades and proxy patterns," 2021. Accessed: 2024-10-04.

[52] Ethereum-Blockchain-Developer, "Upgrade smart contract and smart contract proxies." `https://ethereum-blockchain-developer.com/110-upgrade-smart-contracts/00-project/`, 2021. Accessed: 2024-10-04.

[53] I. Qasse, M. Hamdaqa, and B. Þ. Jónsson, "Smart contract upgradeability on the ethereum blockchain platform: An exploratory study," *arXiv preprint arXiv:2304.06568*, 2023.

[54] Jeiwan, "Upgradeable proxy from scratch," 2021. Accessed: 2024-10-04.

[55] S. Developer, "Eip-2535: Diamond standard." Accessed: 2024-10-04.

[56] Z. Du, H. Cheng, Y. Fu, M. Huang, L. Liu, and Y. Ma, "A four-tier smart contract model with on-chain upgrade," *Security and Communication Networks*, vol. 2023, no. 1, p. 8455894, 2023.

[57] P. Klinger, L. Nguyen, and F. Bodendorf, "Upgradeability concept for collaborative blockchain-based business process execution framework," in *Blockchain–ICBC 2020: Third International Conference, Held as Part of the Services Conference Federation, SCF 2020, Honolulu, HI, USA, September 18-20, 2020, Proceedings 3*, pp. 127–141, Springer, 2020.

[58] . Blockchains, "An introduction to upgradeable smart contracts," 2024. Accessed: 2024-10-04.

[59] Substack, "Answering some diamond questions," 2022. Accessed: 2024-10-04.

[60] E. I. Proposals, "Eip-1538: Transparent contract standard," 2018. Accessed: 2024-10-04.

[61] B. Magnates, "Upgradability patterns in solidity (part 1)." `https://blog.blockmagnates.com/upgradability-patterns-in-solidity-part-1-13e23ce1f144`, 2022. Accessed: 2024-10-04.

[62] R. Verification, "Using foundry to explore upgradeable contracts part 1," 2022. Accessed: 2024-10-04.

[63] S. Boulevard, "Contract upgrade anti-patterns," 2018. Accessed: 2024-10-04.

[64] Flolio, "Proxy smart contract," 2022. Accessed: 2024-10-04.

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

[66] M. Lohr and S. Peldszus, "Maintenance of long-living smart contracts.," in *Software Engineering (Workshops)*, 2020.

[67] CertiK, "Upgradeable proxy contract security best practices," 2022. Accessed: 2024-10-04.

[68] S. A. Amri, L. Aniello, and V. Sassone, "A review of upgradeable smart contract patterns based on openzeppelin technique," *The Journal of The British Blockchain Association*, 2023.

[69] A. Wasnik, "Implementing upgradeable smart contracts using proxy patterns." `https://www.talentica.com/blogs/implementing-upgradeable-smart-contracts-using-proxy-patterns/`, 2024. Accessed: 2024-10-04.

[70] M. Workshop, "Upgradeable smart contracts: Proxy pattern," 2021. Accessed: 2024-10-04.

[71] yos.io, "Upgrading solidity smart contracts," 2018. Accessed: 2024-10-04.

[72] Hacken, "Upgradeable smart contracts," 2023. Accessed: 2024-10-04.

[73] C. M. Pool, "Multiple ways to upgrade a solidity smart contract." `https://cryptomarketpool.com/multiple-ways-to-upgrade-a-solidity-smart-contract/`, 2022. Accessed: 2024-10-04.

[74] O. Oruç, U. Aßmann, and A. Troshani, "Context-based role object pattern with on-chain smart contract programming," in *Konferenzband zum Scientific Track der Blockchain Autumn School 2022*, no. 2, pp. 42–50, Hochschule Mittweida, 2022.

[75] A. Banerjee, "Nuances of using upgradeable smart contracts," 2022. Accessed: 2024-10-04.

[76] P. Collins, "Upgrading your smart contracts - a tutorial & introduction." `https://www.youtube.com/watch?v=bdXJmWajZRY`, 2021. Accessed: 2024-10-04.

[77] E. S. Exchange, "Pros and cons of smart contract upgradable patterns," 2019. Accessed: 2024-10-04.

[78] E. I. Proposals, "Eip-1822: Universal upgradeable proxy standard (uups)," 2019. Accessed: 2024-10-04.

[79] F. W. C. Bloo, "Towards updatable smart contracts," Master's thesis, 2018.

[80] E. I. Proposals, "Eip-1967: Standard proxy storage slots," 2019. Accessed: 2024-10-04.

Chapter 4

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

Authors: Ilham Qasse, Mohammad Hamdaqa, and Björn Þór Jónsson
Status: Submitted to the Demonstrations Track of the 47th International Conference on Software Engineering (ICSE 2025).

Highlights

- Introduces PROXiFY, a bytecode-based tool for detecting and classifying Ethereum proxy contracts without needing source code or transaction data.
- Detects both active and inactive proxies, enhancing transparency across the Ethereum ecosystem.
- Achieves up to 100% precision and high recall rates on various datasets.

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

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

***Abstract*—As Ethereum smart contracts become more complex, upgrading them is essential but challenging due to their immutable nature. Proxy contracts allow upgrades without altering the contract's state, yet current detection methods often depend on source code or transaction data, missing inactive proxies. We introduce PROXiFY, a bytecode-based tool that accurately detects and classifies proxy contracts, including inactive ones, without relying on source code or transaction history. Designed for developers, auditors, and researchers, PROXiFY improves upon existing methods by providing enhanced accuracy and broader coverage in proxy contract detection across both active and dormant Ethereum contracts. A demonstration of PROXiFY can be viewed at https://youtu.be/XwTXIQQbHzM.**

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

I. Introduction

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

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

[1]https://eips.ethereum.org/EIPS/eip-1967
[2]https://eips.ethereum.org/EIPS/eip-1822

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

Current approaches to proxy detection, such as source code or transaction-based analysis, have limitations [7], [11]. Source code analysis is only possible when the code is verified, which applies to a minority of smart contracts on Ethereum [7]. Transaction-based analysis may miss inactive proxies that have not yet been used or upgraded [7], [11]. These gaps highlight the need for a more comprehensive solution.

This paper introduces PROXiFY, a bytecode-based tool aimed at researchers, smart contract auditors, and developers working on Ethereum-based decentralized applications (DApps). PROXiFY detects and classifies proxy contracts without requiring source code or transaction history, addressing the need for accurate classification and enhanced transparency in the smart contract community. By analyzing deployed bytecode, PROXiFY identifies active and inactive proxies, distinguishing between forward and upgradeable proxies. This provides a complete analysis of smart contracts, enhancing the transparency of Ethereum-based DApps [8].

Contributions:

1) PROXiFY reliably detects and classifies proxy contracts, distinguishing between forward and upgradeable proxies, including inactive proxies often missed by other tools.
2) Our evaluation demonstrates that PROXiFY improves precision (by up to 100%) and recall across multiple datasets, correcting misclassifications in previous studies and providing more accurate detection for both active and dormant contracts, without the need for source code or transaction data.

II. Methodology

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

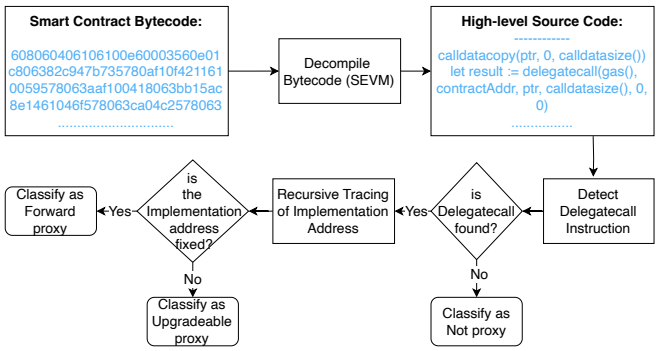

Fig. 1: High-level architecture of PROXiFY,

A. Bytecode Decompilation

The first step involves decompiling the smart contract bytecode to obtain a human-readable representation of the contract's logic. PROXiFY takes the bytecode as input and uses a decompiler to translate it into an intermediate Solidity-like code. We selected SEVM,[3] an EVM decompiler, for its reliability and features. SEVM is lightweight, with no dependencies, and includes an embedded functions and events signature database. It can extract events or function information from bytecode and effectively identify the presence of `delegatecall` instruction.

B. Detection of `Delegatecall` Usage

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

Listing 1: Fallback function performing `delegatecall`

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

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

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

C. Recursive Tracing the Implementation Address

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

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

Listing 2: Modifying the implementation address

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

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

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

Listing 3: Passing the new implementation address to storage

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

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

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

PROXiFY recursively traces these function calls, to determine whether the implementation address is being modified.

D. Classification Logic

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

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

E. Report Generation

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

III. Related Work

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

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

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

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

A. PROXiFY Evaluation

We evaluated PROXiFY, in terms of precision, recall, and F1 score, against two datasets: USCHunt [9] (958 contracts) and Huang et al. [13] dataset (4451 contracts). The USCHunt dataset is based on their tool, which relies on source code analysis for proxy classification. However, the Huang et al. [13] dataset is not derived from a tool but is a manually annotated dataset created by the authors. Unfortunately, other existing tools could not be evaluated as their data or implementations are not publicly available. For the USCHunt dataset, As shown in Table I, PROXiFY achieved high precision and recall (100% and 96.31%, respectively). In contrast, USCHunt exhibited more false positives, largely due to the inclusion of non-main contracts and unreachable code provided in the source code. Specifically, USCHunt identified 155 non-proxy contracts as upgradeable and 169 forward proxies as upgradeable, which led to lower precision and recall for their tool. Other studies [11], [13] also observed this, but they were not available for direct comparison. Nevertheless, we manually assess the disagreement in the evaluation results. We also found that in some cases, PROxiFY fails to decompile the bytecode, which results in false negative cases.

Similarly, in the Huang et al. [13] dataset, as demonstrated in Table II, PROXiFY maintained perfect precision (100%) and near-perfect recall (99.77%). On the other hand, Huang et al. [13] dataset had 438 false positives, mostly due to similar issues with source code analysis, such as contracts that were misclassified due to unreachable functions that modify the code but are not part of the deployed bytecode.

The datasets used for evaluation are available in the project's public repository.[4]

TABLE I: Evaluation Results Based on USCHunt Dataset

Metrics	Precision	Recall	F1 Score
PROXiFY	1.0000	0.9631	0.9812
USCHunt	0.6049	0.9764	0.7470

TABLE II: Evaluation Results Based on Huang et al. [13] Dataset

Metrics	Precision	Recall	F1 Score
PROXiFY	1.0000	0.9977	0.9989
[13]	0.8998	0.9970	0.9459

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

B. Limitations

The primary limitation of PROXiFY lies in the accuracy of decompilation. While SEVM proved reliable in detecting `delegatecall` instructions and did not exhibit timeout issues, the accuracy of the decompiled code directly affects the analysis. In some cases, we encountered failures when automating the feeding of multiple bytecodes to evaluate the tool. However, there were no failures when these bytecodes were tested individually using the PROXiFY webpage. This suggests that automation may introduce challenges that do not occur during individual analysis. Other decompilers we tested, such as Heimdall[5] and Panoramix[6], either failed to detect delegatecall or experienced timeouts, respectively. Therefore, while SEVM addresses some of these issues, decompilation remains a critical factor that can impact the tool's effectiveness.

IV. Applications of PROXiFY

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

First, researchers studying smart contracts' evolution or security implications can utilize PROXiFY to analyze datasets and classify proxies without requiring source code. By detecting active and inactive proxies, PROXiFY enables researchers to conduct empirical studies on upgradeability mechanisms, offering new insights into how smart contracts evolve and maintain state over time. This tool has already been used in our empirical study [7], which analyzed the prevalence of upgradeable smart contracts on Ethereum.

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

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

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

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

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

V. Conclusion

In this paper, we presented PROXiFY, a bytecode-based tool for detecting and classifying proxy contracts within Ethereum smart contracts. As the need for contract upgrades becomes increasingly essential in evolving decentralized applications, traditional detection methods are often insufficient, especially in identifying inactive or unverified proxies. By focusing on bytecode-level analysis, PROXiFY addresses key limitations of existing tools, enabling more accurate detection of both forward and upgradeable proxies without requiring source code or transaction history. Through an extensive evaluation based on precision, recall, and F1 score, PROXiFY demonstrated higher performance across all three metrics compared to existing available approaches. This makes PROXiFY a valuable tool for developers, auditors, and researchers working within the Ethereum ecosystem. Future work includes optimizing the decompilation process for faster large-scale analysis.

References

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

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

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

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

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

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

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

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

[9] W. E. Bodell III, S. Meisami, and Y. Duan, "Proxy hunting: Understanding and characterizing proxy-based upgradeable smart contracts in blockchains," in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 1829–1846, 2023.

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

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

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

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

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

Chapter 5

Immutable in Principle, Upgradeable by Design: Exploratory Study of Smart Contract Upgradeability

Authors: Ilham Qasse, Mohammad Hamdaqa, and Björn Þór Jónsson
Status: Second-round revisions submitted to Empirical Software Engineering (EMSE) journal. Initially accepted as a Registered Report in the 20th International Conference on Mining Software Repositories (MSR).

Highlights

- Conducts an empirical study of smart contract upgradeability on Ethereum, analyzing over 44 million contracts.
- Finds that only 3% of contracts support upgradeability, with only 0.34% actively upgraded, indicating conservative upgrade behavior by developers.
- Identifies security and feature enhancements as the main motivations for contract upgrades, especially in open-source contracts.
- Observes that user activity is influenced by factors beyond the number of contract versions, indicating complexity in user engagement with upgradeable contracts.
- Develops a novel dataset and classification for tracking and analyzing smart contract upgrades on the Ethereum blockchain.

Noname manuscript No.
(will be inserted by the editor)

Immutable in Principle, Upgradeable by Design: Exploratory Study of Smart Contract Upgradeability

Ilham Qasse · Mohammad Hamdaqa · Björn Þór Jónsson

Received: 01/12/2023

Abstract Smart contracts, known for their immutable nature to ensure trust via automated enforcement, have evolved to necessitate upgradeability due to unforeseen vulnerabilities and the need for feature enhancements post-deployment. This inherent contradiction between immutability and the necessity for modifications has prompted the development of upgradeable smart contracts. These contracts are immutable in principle yet upgradable by design, allowing for future updates without altering the underlying data or state, thus preserving the contract's original intent while accommodating necessary improvements.

This empirical study aims to bridge the gap in understanding the practical application and implications of upgradeable smart contracts on the Ethereum blockchain. By introducing a comprehensive dataset that catalogs the versions and evolutionary trajectories of smart contracts, the research explores several key dimensions: the prevalence and adoption patterns of upgrade mechanisms, the likelihood and actual occurrences of contract upgrades, the nature of modifications post-upgrade, and their impact on user engagement and contract activity. Through a detailed empirical analysis, this study systematically identifies upgradeable contracts and examines their upgrade history to uncover trends, preferences, and the practical challenges associated with applying such modifications.

The empirical evidence gathered from the analysis of over 44 million contracts shows that a mere 3% embody upgradeable characteristics, with only 0.34% of these undergoing subsequent upgrades. This finding underscores a

I. Qasse
Reykjavik University, Reykjavík, Iceland
E-mail: ilham20@ru.is

M. Hamdaqa
École Polytechnique de Montréal, Montréal, Canada

B. Þ. Jónsson
Reykjavik University, Reykjavík, Iceland

cautious approach by developers towards contract modifications, possibly due to the complexity of upgrade processes or a preference for maintaining the original contract stability. Furthermore, the study demonstrates that upgrades are predominantly aimed at feature enhancement and vulnerability mitigation, particularly when the contracts' source codes are accessible. However, the relationship between contract upgrades and user activity is intricate, suggesting that additional factors significantly affect the utilization of smart contracts beyond their mere functional evolution.

Keywords Smart Contract · Ethereum · Upgradeability · Proxy Contract · Solidity · Empirical Study · Dataset

1 Introduction

A smart contract is computer code deployed onto the blockchain that automates the enforcement, monitoring, and execution of agreements when predetermined conditions are met (Wang et al., 2019; Buterin et al., 2014; Sayeed, Marco-Gisbert, & Caira, 2020; Mohanta, Panda, & Jena, 2018). While initially envisioned to be immutable to ensure trust and reliability, acting as an unalterable agreement between parties, the practical application of smart contracts has revealed the necessity for upgradeability (He et al., 2020; Qian et al., 2022; Samreen & Alalfi, 2021; Rodler, Li, Karame, & Davi, 2021). This realization stems from the dynamic nature of software, where vulnerabilities and bugs discovered post-deployment necessitate updates to ensure security and functionality (He et al., 2020; Qian et al., 2022; Samreen & Alalfi, 2021; Sayeed et al., 2020; Rodler et al., 2021). Thus, despite their foundational principle of immutability, the necessity for smart contracts to be upgradeable has become evident, highlighting the importance of modification capabilities to enhance functionality and address security concerns (Antonino, Ferreira, Sampaio, & Roscoe, 2022; Salehi, Clark, & Mannan, 2022; Sayeed et al., 2020; Rodler et al., 2021).

In contrast to traditional software, where a software upgrade can introduce changes to any part of the software, a smart contract upgrade refers to a process of arbitrarily modifying the contract code logic while maintaining its stored data or state (Antonino et al., 2022; Salehi et al., 2022; Bodell III, Meisami, & Duan, 2023). This requirement points to a key difference between smart contract upgrades and conventional software updates, where data preservation is less critical. An upgradeable smart contract is designed for possible future modifications by leveraging patterns that enable updates to its logic or features; hence, not all deployed smart contracts are upgradeable. Initially, the community relied on data migration to upgrade contracts, a method that, although effective in updating the contract's logic, did not align with the fundamental principle of preserving the contract's original state. This costly and complex approach led to the exploration of more efficient upgrading methods (Antonino et al., 2022; Salehi et al., 2022; Bodell III et al., 2023).

In response to the limitations of the data migration approach, the community has introduced upgrading techniques, such as the proxy patterns [1,2,3], which allows for modifying the contract's logic without affecting its state. These advancements have facilitated the creation of upgradeable smart contracts, yet there remains a gap in understanding how these approaches, especially the proxy pattern, are applied and their impact. The exploration of upgradeable smart contracts, particularly in assessing the prevalence of such contracts, how likely they are to be upgraded, the practical applications of upgrade patterns, and the impact of these upgrades on user engagement and trust, is still in its nascent stages.

This empirical study aims to provide an empirical perspective on smart contract upgradeability, addressing the research void with data-driven insights. It focuses on smart contracts deployed on the Ethereum blockchain due to its open-source nature and active community engagement in discussions and implementations related to smart contract upgradeability. The study collects existing smart contracts, focusing on their upgradeability features, to create a comprehensive dataset that catalogs smart contracts' versions and their evolution. Moreover, it analyzes the patterns of upgrades through the lens of evolution lineage (different versions of deployed smart contracts) to answer the following Research Questions(RQs):

– **RQ1: How prevalent are upgrade patterns in smart contracts?** Although the smart contract community proposed many approaches to upgrading smart contracts, there is limited understanding of their adoption by developers and the prevalence of these methods in practice. Our research aims to fill this gap by designing policies that can be used as rules to identify upgradeable contracts and the approaches for their upgrades. These policies leverage regular expressions and code structure analysis of decompiled bytecode, focusing on the proxy pattern family. We concentrate on seven community-endorsed standards to analyze the adoption of upgradeable mechanisms systematically. The scope of this question includes all smart contracts, allowing us to assess the overall prevalence and adoption of different upgradeable proxy patterns across the entire smart contract ecosystem.
  Our findings indicate that while upgradeable proxy contracts represent a small percentage of the total smart contracts analyzed, there is a clear preference among developers for certain upgrade patterns. This suggests a discernible trend in adopting specific standards, which has implications for the design and implementation of upgradeable smart contracts. Additionally, our research identifies previously unrecognized types of upgradeable contracts that diverge from established standards, suggesting innovative or customized approaches to upgradeability within the community.

[1] `https://eips.ethereum.org/EIPS/eip-1822`
[2] `https://eips.ethereum.org/EIPS/eip-2535`
[3] `https://eips.ethereum.org/EIPS/eip-1967`

The significance of this research lies in its contribution to understanding the current landscape of smart contract upgradeability. By understanding this landscape, developers can identify which upgrade patterns are most commonly used, providing a benchmark for their own practices. Platform providers can use these insights to better support popular upgrade approaches, while security auditors can focus on securing the most widely adopted upgrade standards.

– **RQ2: How likely is an upgradeable contract to be upgraded?**
Designing a contract to be upgradeable does not ensure it will be upgraded throughout its lifecycle. This question quantifies the extent to which contracts identified as upgradeable in our preceding analysis (RQ1) undergo actual upgrades. This exploration is critical for discerning the gap between the theoretical potential for contract evolution and the practical application of these upgrade mechanisms. Understanding this gap is helpful for different stakeholders. For developers and project managers, it shows whether adding upgradeability is worth the effort. Investors and regulators can use this information to judge how stable or flexible a project might be based on its upgrade patterns.
To address this, we trace the historical versions of smart contracts by analyzing logs and events associated with upgradeable contracts. This data is sourced from the Ethereum ETL dataset, focusing on upgrade requests and the subsequent new version addresses. We specifically look for emitted events that signal an upgrade and trace them to compile a comprehensive list of smart contract versions and their addresses. Hence, this research question focuses on upgradeable proxy contracts that emit events, irrespective of source code availability, to determine the frequency and likelihood of actual upgrades. This approach enables us to identify when upgrades have been implemented systematically and to shed light on the real-world use of upgrade capabilities.
Our findings reveal a notable restraint in the actual upgrading of contracts, with a minority of upgradeable contracts experiencing changes. This highlights a cautious approach to applying upgrades in practice, contrasting with the theoretical capability for frequent modifications.

– **RQ3: What changes occur in smart contracts post-upgrade?**
Traditional software maintenance is well-established and classified into types such as corrective (fixing bugs), preventive (updating documentation, refactoring code), adaptive (updating for new environments), and perfective (adding new features, optimizing performance) (Williams & Carver, 2010; J. Chen, Xia, Lo, Grundy, & Yang, 2021; Chapin, Hale, Khan, Ramil, & Tan, 2001). These classifications help in understanding the various motivations behind software updates and modifications. However, there is limited clarity on the application of these practices to smart contracts (J. Chen et al., 2021). In this research question, we aim to explore whether smart contract upgrades reflect these conventional maintenance activities, focusing on changes like bug fixes, feature modification, and efficiency improvements. Understanding these changes can benefit various stakeholders. De-

velopers and project managers can better understand what types of upgrades are most common, such as security fixes or feature additions, and prioritize future improvements accordingly. Security auditors can focus on ensuring vulnerabilities are properly addressed during upgrades, particularly in security-critical areas. For investors and users, the nature of upgrades provides insights into project maintenance. Frequent security updates may signal a proactive approach to risk management, while ongoing feature modifications could indicate evolving project requirements.
To answer this question, we analyze the verified smart contract versions extracted from RQ2, which have available source code. This analysis involves a comprehensive review of code changes between versions, utilizing Git diff for code comparison and SmartBugs for security vulnerability detection. Additionally, we evaluate gas consumption alterations to identify instances of preventive maintenance through optimization efforts.
Through this process, we categorize the post-upgrade changes into fixing vulnerabilities, feature modification (adding or deleting features), gas optimization, or a residual "other" category for modifications that defy these conventional classifications. The results show a significant portion of upgrades is categorized under "other," indicating a variety of motivations beyond the primary ones identified.

– **RQ4: How does smart contract upgrading impact the activity level of the contract?**
Smart contract upgradeability may lead to trust concerns among users, given the value placed on their immutability. The introduction of upgrades, altering a contract's internal logic or behavior, could necessitate user adaption to interface or transaction gas consumption changes. Frequent upgrades or those not communicated effectively risk diminishing user trust and reducing contract activity.
Evaluating the impact of upgrades on smart contract usage is important for developers and project managers. This information helps them determine whether upgrades encourage or discourage user interaction and decide if frequent updates are beneficial or harmful to the contract's overall activity.
To study this, we focus on all smart contract versions identified in RQ2, particularly upgradeable proxies that emit events, to analyze how upgrades affect contract activity levels. We employ a detailed analysis of transaction volumes pre- and post-upgrade, extracted from Etherscan, adjusting for the contract versions' lifespan to ensure accurate assessment. This involves regression analysis to discern the effects of upgrades on contract engagement, providing insights into whether and how upgrades influence user interactions with smart contracts.
The results demonstrate that most contracts (90.16%) only recorded a single transaction, typically linked to their creation, indicating a prevalence of non-active contracts. Moreover, both linear and non-linear models exhibit limitations in capturing the relationship between upgrades and activity, suggesting that factors beyond lifespan and version number influence contract activity.

Our findings offer new insights into the nature and practical application of upgradeable contracts; a family of smart contracts that are *immutable in principle yet upgradable by design*. In summary, the study's contributions are:

1. We introduce an extensive dataset of smart contract versions, emphasizing their evolution. This dataset provides a valuable foundation for future research on smart contract upgradeability, offering insights into the development and maintenance of these contracts. The dataset, along with the project's source code, is publicly available in the project repository.[4]
2. Through an in-depth empirical analysis, we investigate the prevalence of upgrade patterns in smart contracts (RQ1), identifying specific trends and preferences in adopting upgrade mechanisms. This analysis helps clarify the landscape of smart contract upgradeability, showcasing the diversity of approaches within the community.
3. We quantify the likelihood of upgradeable contracts being actually upgraded (RQ2), highlighting a cautious approach by developers towards modifying smart contracts. This contributes to understanding the gap between the potential for upgrades and their real-world application.
4. Our study delves into the nature of changes implemented through smart contract upgrades (RQ3), examining how these align with or diverge from traditional software maintenance practices. By categorizing these changes, we provide insights into the motivations behind contract upgrades.
5. We analyze the impact of smart contract upgrades on user interaction and activity levels (RQ4), discovering that upgrades do not necessarily lead to increased contract usage. This finding suggests that other factors, beyond the technical aspects of upgrades, significantly influence user engagement with smart contracts.

The remainder of this paper is organized as follows. Section 2 presents the principles of smart contracts and their upgradability. Section 3 discusses the research methodology utilised to answer the research questions. We present the results and discussion in Sections 4 and 5, respectively. Next, Section 6 presents the related work while Section 7 discusses threats to research validity. Finally, the paper is concluded in Section 8.

2 Background

This section introduces concepts related to this study's research questions, by providing an overview of smart contracts and their upgradeability.

2.1 Smart Contract

Smart contracts are computer programs that auto-execute agreements when certain conditions are satisfied. Smart contracts are mostly deployed on a

[4] `https://drive.google.com/drive/folders/1xlY7suki5TqiOF7mipKspfBbTb888-iz?usp=sharing`

blockchain network, enabling the execution of the contract to be secure and transparent (Wang et al., 2019; Buterin et al., 2014; Sayeed et al., 2020; Mohanta et al., 2018; Wang et al., 2018). Many blockchain platforms support smart contracts, such as Hyperledger Fabric (Androulaki et al., 2018), Corda (Brown, Carlyle, Grigg, & Hearn, 2016), and Ethereum (Buterin et al., 2013). This paper focuses on the smart contracts deployed on Ethereum due to the platform's open-source architecture and community. Ethereum is an open-source platform, meaning the underlying code and protocols are publicly available. Moreover, it has a large active community that supports researchers and developers with smart contracts.

Ethereum runs its deployed smart contracts on the Ethereum Virtual Machine (EVM), a low-level stack machine that executes the compiled bytecode of the smart contract (Buterin et al., 2013). Each operation in Ethereum requires a certain amount of computational effort, measured by gas. Gas is required for every operation in Ethereum, whether the operation is a transaction or the execution of a smart contract instruction. Some instructions are gas-intensive, such as instructions that utilize replicated storage. Gas metering prevents the sender from wasting computational power in executing unnecessary computation-intensive transactions. Moreover, it limits the number of instructions a transaction can execute, preventing non-terminating executions and DoS attacks.

Smart contracts in Ethereum are developed in many languages, such as Solidity, Vyper, and Bamboo. In this study, we target Solidity smart contracts, the most popular smart contract programming language (Clack, Bakshi, & Braine, 2016). The Solidity language is an object-oriented language deployed on the EVM. In addition to Ethereum, several blockchain platforms support Solidity, including Quorum, Hyperledger Burrow, and Hyperledger Besu. Solidity has syntax similar to C and JavaScript, but it includes several unique concepts specific to smart contracts, including: visibility of function modifiers; internal, external, view; emitted events; and smart contract-specific operations such as self-destruct and revert.

We focus on Solidity contracts as there are more than 44 million Solidity contracts deployed on the Ethereum network.

2.2 Smart Contract Upgradeability

In smart contracts, upgradeability refers to the process of modifying smart contract code after deployment while maintaining the contract data and state (Antonino et al., 2022; Salehi et al., 2022). Upgrading smart contracts has two benefits: (i) it provides a mechanism to improve contract security by fixing security issues and bugs discovered post-deployment, and (ii) it enables developers to add new features and functionality over time (Antonino et al., 2022; Salehi et al., 2022).

In the context of smart contracts, upgradeability and mutability (ability to change) are different. Since smart contracts are immutable in principle,

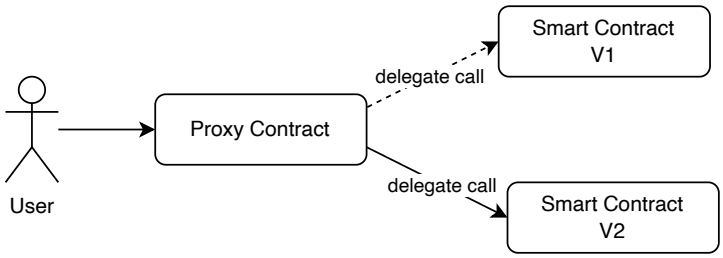

Fig. 1 Proxy pattern for upgrading smart contract

their code cannot be changed once deployed. However, the smart contract community proposed several mechanisms to upgrade the contract, such as deploying a new smart contract and directing user requests to the new one instead of the previous one.

The proxy pattern is a well-known method where users interact with a proxy contract, not directly with the business logic contract. The proxy contract holds data and forwards user requests to the designated smart contract version. When an upgrade is necessary, a new contract is deployed, and its address is updated in the proxy contract as the target version, as shown in Figure 1. This approach allows for the seamless transition between contract versions without disrupting the user experience. Other variations of the proxy pattern include Diamond Proxy, Universal Upgradeable Proxy Standard (UUPS), EIP-1967, and OpenZeppelin Inherited Storage. The Diamond Proxy allows for multiple logic contracts (called facets) to be used simultaneously, providing greater flexibility and modularity by enabling upgrades to individual facets rather than the entire contract. UUPS simplifies the upgrade process by embedding the upgrade logic directly into the implementation contract instead of the proxy, reducing the proxy's complexity. EIP-1967 standardizes the storage slots used by proxy contracts, ensuring a more consistent and secure approach to managing upgradeable contracts. Finally, OpenZeppelin Inherited Storage builds on the proxy concept by integrating inherited storage slots, allowing flexible upgrades without altering the contract's state storage structure. The data separation pattern is another upgrading approach. It involves two contracts: one for storage and the other for business logic. Unlike the proxy pattern, users interact directly with the business logic contract, which interacts with the storage contract. This separation ensures that data remains intact and secure, even as the logic contract is upgraded or modified. The storage contract is a consistent data layer accessible only by authorized contract versions, ensuring data integrity and security. However, it introduces inefficiencies due to cross-contract calls between the storage and logic contracts, leading to increased gas costs and added latency. This makes the Data Separation approach less appealing for projects requiring frequent interactions, as the overhead can accumulate significantly.

Additionally, the Data Migration method provides an alternative way to manage upgrades by migrating data from an old contract to a newly deployed

contract. This method allows for extensive changes to both the logic and structure of the contract but at the cost of complexity. Migrating data can be error-prone and time-consuming, requiring manual interventions to ensure data integrity throughout the transition. The high risk of errors during migration and the manual efforts involved make this method less favorable for most projects, especially those needing consistent and reliable operations.

The Strategy Pattern represents a more modular approach, focusing on breaking down different functionalities or algorithms into interchangeable components. This allows developers to swap individual contract components without affecting the rest of the contract's logic. However, the Strategy Pattern is more aligned with general contract design principles and is less about facilitating upgradeability. As a result, it is less commonly used for managing smart contract upgrades specifically. Table 1 provides a detailed comparison of these approaches in terms of structure, efficiency, security, standardization, adoption, and the upgrade process.

Table 1 Comparison Between Smart Contract Upgrade Approaches

Feature	Proxy Patterns	Data Separation	Data Migration	Strategy Pattern
Structure	Proxy holds data, forwards logic calls via `delegatecall`	Two separate contracts: one for logic and one for storage	Migrates data to a new contract version	Uses interfaces to swap specific logic without impacting entire contract
Efficiency	Lower gas costs due to internal `delegatecall`	High gas costs from cross-contract calls	Inefficient due to manual data migration	Efficient for swapping specific logic, but can add complexity
Security	Risks from function clashes or admin privileges depending on proxy type	Cross-contract interactions increase security risks	High security risk during manual data transfer	More complex to manage, only specific functions can be upgraded
Standardization	Standardized by EIPs, widely used in Ethereum	No standardization, diverse implementations	Custom implementation, no formal standard	Not standardized, lacks formal guidelines
Adoption	Widely adopted in Ethereum projects	Complex, requires careful coordination between the new version storages	-	-
Upgrade Process	Streamlined upgrades via proxy contract	Complex, requires careful coordination between version's storage	Complex data transfer process	Upgrades only specific logic modules, not the full contract

As shown in Table 1, proxy patterns are the most widely adopted solutions for managing upgrades in the Ethereum ecosystem. They allow developers to upgrade the logic of smart contracts while maintaining their data, making them well-suited for projects that require frequent updates or changes. In contrast, the Data Separation and Data Migration methods, although capable of preserving data integrity or enabling substantial changes, introduce significant inefficiencies and risks. These methods involve complex processes, such as cross-contract interactions or manual data migration, which increase costs and the potential for errors. As a result, their adoption has been minimal compared to the streamlined and standardized proxy patterns.

In this study, we focus on the upgrading approaches introduced by the Ethereum community, particularly proxy-based solutions. We apply inclusion and exclusion criteria to select suitable approaches for this empirical study, as discussed in Section 3.3.

3 Study Methodology

This study aims to empirically mine the upgradeable smart contract patterns from the deployed smart contracts to provide evidence of their prevalence and usage scenarios. Figure 2 illustrates an overview of our study methodology, including data collection, data preprocessing, identifying upgradeable smart contracts, and analyzing smart contracts version evolution and their impact on the contract's activity level.

3.1 Data Sources

In our study, we integrate data from two sources, which are Ethereum ETL,[5] and Etherscan.[6] Ethereum ETL is a public Ethereum data explorer that enables users to explore blockchain data such as blocks and transactions. All the data is published as a Google BigQuery dataset.[7] The Ethereum ETL dataset includes details about blocks, transactions, and smart contracts. Ethereum ETL configures nodes to synchronize data with the Ethereum blockchain, where these data are updated regularly. We extract from the dataset smart contract metadata such as smart contract address, bytecode, and the contract creator address.

To obtain more details about the smart contract, such as the contract's source code and the contract's activity level, we use Etherscan, an Ethereum explorer. The data on Etherscan is updated in real-time, as it syncs with nodes configured on the Ethereum network. Based on each contract's address extracted from Ethereum ETL, we query Etherscan for the contract address and extract the contract's source code and activity level, if available. There are unverified smart contracts in Etherscan, which means that the developer didn't provide the source code of the smart contract. In this study, we do not consider these smart contracts.

3.2 Data Collection and Preprocessing

To answer the research questions, we preprocess the collected smart contracts data as follows:

– Remove comments and white spaces: We used a custom Python script to remove comments and whitespace from the smart contracts. This step reduces the size of the collected data and ensures a consistent structure and format across all contracts.

[5] `https://ethereum-etl.readthedocs.io/en/latest/`

[6] `https://etherscan.io`

[7] `https://bigquery.cloud.google.com/dataset/bigquery-public-data:crypto_ethereum`

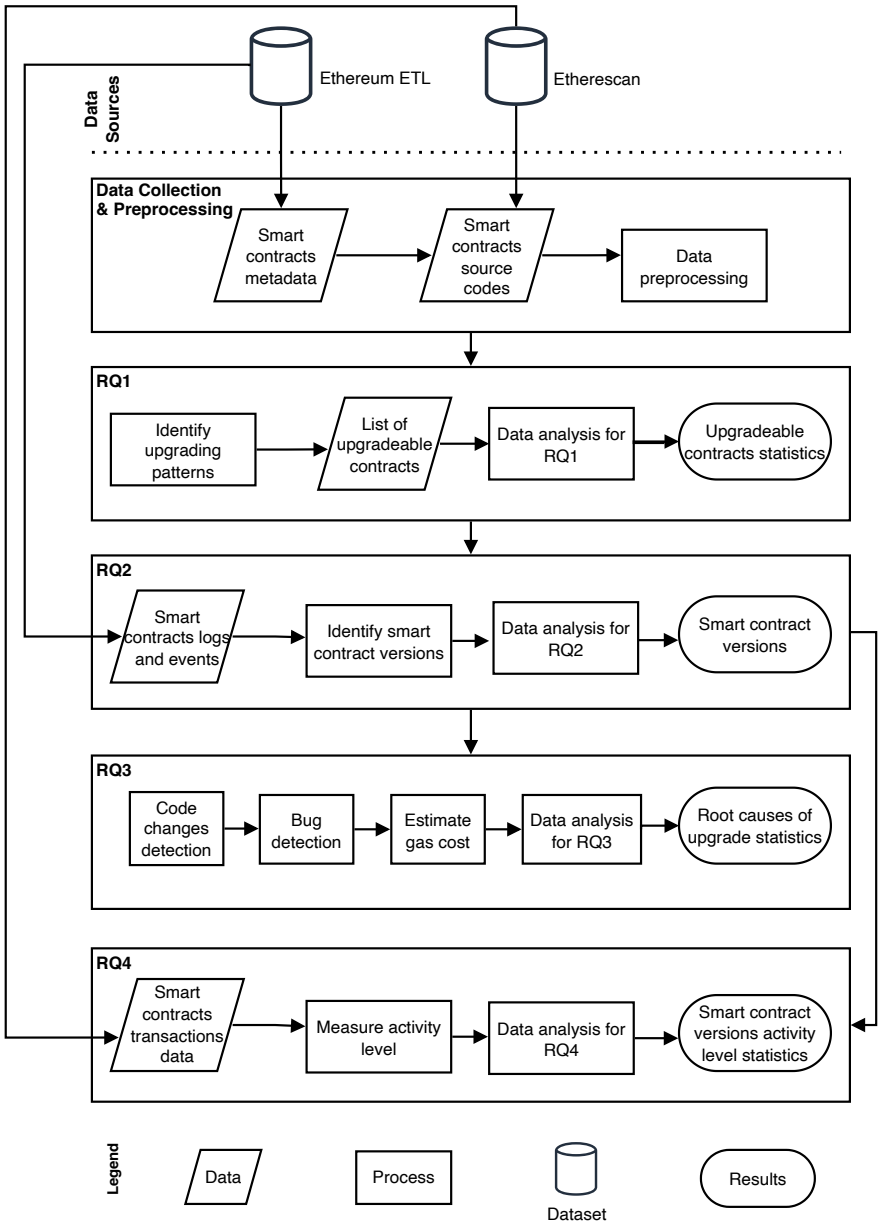

Fig. 2 Overview of the study methodology

– Reformat multi-file smart contracts: Smart contracts can be coded using one or multiple files. As we are comparing different versions of the smart contracts, it is essential to have each smart contract in a single file. For this purpose, we merge files and remove imports within files for multi-file smart contracts. To handle potential differences in file ordering from Etherscan, we included the file names and sorted them alphabetically before using git diff for content-based comparison.

The data collection spans from January 2018 to July 2023. This period aligns with the initial introduction of upgradeable contract standards and practices, such as the unstructured storage pattern in OpenZeppelin's early upgradeable contracts and the first submissions of the EIP-1822 and EIP-1967 standards. We collected 44M deployed smart contracts. Of these, 504,853 are verified contracts, based on data from Etherscan during the same period. For each smart contract, we collected the following metadata: the smart contract address, the contract creator address, the timestamp, the compiler version, and the Solidity version. Moreover, we collected the contract source code, where available, and the number of received transactions from Etherscan. The focus and scope of the study vary across the research questions: RQ1 analyzes the entire dataset to identify upgradeable proxy contracts; RQ2 focuses on the subset of upgradeable proxy contracts that emit events, using data from EthereumETL; RQ3 examines the verified smart contract versions (from the 504,853 contracts) based on Etherscan data to analyze post-upgrade changes; and RQ4 investigates transaction details from all smart contract versions identified in RQ2 to assess the impact of upgrades on contract activity levels.

3.3 Identifying Upgradeable Smart Contracts

To answer RQ1, in our study on the prevalence of upgradeable smart contracts, we specifically focus on analyzing patterns that are widely recognized within the Ethereum,[8] and OpenZeppelin[9] communities. We focus our analysis on patterns that offer clear, identifiable code and storage structures, enabling straightforward identification and differentiation. We exclude approaches that do not align with our criteria for systematic analysis. This includes ad-hoc upgrading methods such as Contract Migration, which often depend on manual interventions and lack clear distinguishable patterns in contract code. Similarly, we excluded the Data Separation approach due to its minimal prevalence and adoption by the community, representing only 0.0007% of the collected data (Salehi et al., 2022). Furthermore, we also omitted generic patterns focused more on design principles, like the Strategy Pattern, for their lack of direct relevance to upgradeable smart contracts.[10] These exclusions are based

[8] https://ethereum.org/en/developers/docs/smart-contracts/upgrading/

[9] https://blog.openzeppelin.com/the-state-of-smart-contract-upgrades

[10] https://blog.openzeppelin.com/the-state-of-smart-contract-upgrades#strategy-pattern

Listing 1 Upgradeable proxy contract example

```
pragma solidity ^0.8.0;

contract Proxy {
    address public implementation;

    constructor() public {
        implementation = address(new Implementation());
    }

    function upgradeTo(address _implementation) public {
        require(msg.sender == msg.sender, "Only the owner can ↩
            upgrade the contract");
        implementation = _implementation;
    }

    function execute(bytes memory _data) public {
        require(implementation != address(0), "Implementation ↩
            contract not set");
        (bool success, bytes memory returnData) = address(↩
            implementation).delegatecall(_data);
        require(success, "Execution failed");
    }
}
```

on their limited relevance to our research objective that aims to shed light on common practices of upgradeability in smart contracts.

Accordingly, this work focuses on approaches that exhibit unique code features that enable us to establish differentiation criteria from other approaches (patterns). Notably, we concentrate on the family of proxy patterns, which exemplify the current best practices in smart contract upgradeability. These approaches are identifiable by specific features in their code, allowing us to establish policies that differentiate them from other approaches.

To detect upgradeable proxy contracts, we identify a set of policies that distinguish them from other techniques. These policies include a combination of regular expressions applied to the decompiled bytecode and analysis of the code structure specific to the upgrade pattern. For example, the presence of *delegatecall* is indicative of a proxy pattern that can be detected using a simple regular expression on the decompiled bytecode. However, not all proxy contracts are upgradeable; forward proxy contracts direct requests to other contracts without enabling upgrades. To distinguish these two types, it is necessary to analyze the decompiled code structure to trace how the implementation address is managed. Our detection process, as illustrated in Figure 3, begins by decompiling the bytecode of smart contracts using the Dedaub decompiler [11] to obtain a more readable form. We then analyze the decompiled bytecode to detect the presence of the *delegatecall* instruction. If delegatecall

[11] `https://app.dedaub.com/decompile`

Listing 2 Forward proxy contract example

```
pragma solidity ^0.8.0;

contract Proxy {
    address public implementation;

    constructor(address _implementation) public {
        implementation = _implementation;
    }

    function execute(bytes memory _data) public {
        require(implementation != address(0), "Implementation ↩
            contract not set");
        (bool success, bytes memory returnData) = address(↩
            implementation).delegatecall(_data);
        require(success, "Execution failed");
    }
}
```

is detected, it indicates the contract is a proxy. We then examine how the target address for *delegatecall* is managed, whether hardcoded or passed as a variable.

If the implementation address is hardcoded directly in the *delegatecall*, the contract is classified as a forward proxy. However, if the implementation address is passed as a variable, as shown in Listings 1 and 2, where the *delegatecall* uses the *implementation* variable, we trace this variable throughout the contract. We then determine whether it can be modified by other functions within the contract. In Listing 1, the *upgradeTo* function allows modification of the implementation variable, indicating that the contract is upgradeable.

If no such modification function exists, as shown in Listing 2, and the implementation variable remains constant, we classify the contract as a forward proxy since the implementation address cannot be modified. While Listings1 and 2 are provided in source code for conceptual clarity, our detection is conducted at the decompiled bytecode level, which may not appear exactly as shown.

If the proxy contract is identified as upgradeable, we proceed to further classify the type of upgrade mechanism it uses. Our classification covers a range of key upgradeable proxy standards proposed by the Ethereum Improvement Proposals (EIP),[12] and OpenZeppelin[13] communities. These standards detail the technical requirements necessary for upgradeable contracts and establish a unified terminology for our analysis. Moreover, this allows us to identify and describe the different upgradeability methods systematically. By focusing on these patterns and standards, our study seeks to provide a detailed overview

[12] https://eips.ethereum.org

[13] https://www.openzeppelin.com

of current practices in upgradeable proxy contract approaches. Accordingly, this paper concentrates on the following approaches:

- EIP-897: DelegateProxy (Araoz & Izquierdo, 2018) is a standard interface for interacting with various proxy types. It explicitly differentiates between two proxy types: forwarder proxies, which primarily act as relays to other contracts, and upgradeability proxies, designed to enable the evolution of contract logic over time. This standard provides developers and contract owners with a structured way to implement upgradeable contracts, minimizing the risks associated with custom implementations. Furthermore, block explorers like Etherscan benefit by being able to show both the actual ABI and code for the contract, improving transparency for users. However, stakeholders such as developers and auditors should be aware of potential issues like function selector collisions, which could lead to unintended behavior during upgrades, necessitating careful management.
- ERC-1538: Transparent Contract Standard (Mudge, 2018) introduces a transparent way to manage smart contract upgrades. It focuses on enabling the addition, removal, and modification of functions within a contract in a clear and traceable manner. This standard builds trust by providing a clear record of changes, benefitting users and investors who can track the contract's history. Security auditors also benefit from the traceability, as it simplifies the verification of modifications over time.
- EIP-1967: Proxy Storage Slots (Palladino, Giordano, & Croubois, 2019) addresses the necessity of a standardized approach for tracking proxy contracts and their corresponding logic contract addresses. The lack of a common interface for accessing this information had previously been a barrier to developing tooling for proxies. EIP-1967 responds to this need by specifying distinct storage slots within a proxy's storage, aiming to streamline the identification and management of the logic contract and the proxy's administrative control. EIP-1967 introduces two distinct types of storage slots for proxies: Proxy Logic Storage Slots and Proxy Beacon Storage Slots. The Proxy Logic Storage Slots are utilized to store the address of the logic contract that the proxy delegates call to. This standardization facilitates the easy location and updating of the logic contract in a proxy setup. On the other hand, Proxy Beacon Storage Slots are employed in Beacon Proxy patterns. They store the address of a beacon contract, which in turn points to the logic contract. This method allows multiple proxy contracts to be upgraded by updating a single beacon contract. Developers and contract owners benefit from this streamlined upgrade process, as it simplifies the interaction between proxy and logic contracts and reduces the chances of mistakes during upgrades. Auditors benefit too, as the clearly defined structure aids in inspecting upgrade procedures. For users and tool developers, having a common interface allows block explorers to present proxy and logic contract information, improving the overall user experience. However, stakeholders should remain cautious about the administra-

tive privileges associated with proxies, as mismanagement could introduce security vulnerabilities.
- EIP-1822: Universal Upgradeable Proxy Standard (UUPS) (Barros & Gallagher, 2019) enhances the developer experience by incorporating upgradeability directly into the logic contract. Unlike other upgradeability patterns where the proxy contract triggers updates, the UUPS allows the logic contract to initiate its upgrade. This inversion of control not only streamlines the upgrade process but also provides contract owners the flexibility to remove upgradeability when desired, which can be appealing to investors seeking stable environments. Standardizing bytecode verification helps auditors ensure the integrity of upgrades. However, the centralization of upgrade control increases the importance of thorough audits, as vulnerabilities in the logic contract could pose significant risks.
- EIP-2535: Diamonds or Multi-Facet Proxy ((@mudgen), 2020) presents a flexible, efficient approach for smart contract development. Unlike traditional single logic contract proxies, ERC-2535 allows a smart contract to use multiple logic contracts, known as facets, enabling a more modular and extensible structure. This design allows developers to add, replace, or remove functionalities (facets) without deploying a new contract. Contract owners, too, benefit from the incremental evolution of the contract, reducing user disruption. Security auditors also find it advantageous, as they can focus on individual facets during their audits. However, the complexity of managing multiple facets can pose challenges, and stakeholders must ensure proper coordination between facets to prevent functional or security issues.
- The OpenZeppelin Inherited Storage Proxy Pattern (OpenZeppelin, 2018) allows for upgrading a contract's behavior while preserving the storage structure of previous versions. OpenZeppelin, a pioneer in proxy contract development since 2017, has contributed significantly to developing patterns like the Inherited Storage Proxy, which supports upgradeable contracts without losing state. The core idea is that each new version of the logic contract follows the storage layout of the previous one, ensuring that upgrades do not disrupt the existing state. Developers and contract owners benefit from this approach, as it maintains storage continuity across upgrades, reducing the risk of state corruption. Auditors also benefit from the consistency in storage structure, simplifying their verification processes during upgrades. However, developers must be cautious when modifying the storage structure, as only append-only changes are safe, limiting the flexibility of some upgrades. This limitation may require careful planning to ensure compatibility across future contract versions.

We designed policies derived from the official standards specifications and utilized the Evm-Proxy-Identification tool,[14] a well-known tool for detecting proxy contracts. However, due to the API constraints of the tool, we developed an independent proxy detection mechanism. This mechanism goes beyond sim-

[14] `https://github.com/CaiJiJi/evm-proxy-detection/`

Table 2 Proxy types and their unique identifiers

Label	Proxy Type	Storage Slot/Identifier
A	EIP-897: DelegateProxy	ox5c60da1b0000000000000000000000000000000000000000000000000000000000000000
B	ERC-1538 : Transparent Contract Standard	0x61455567 (Interface Identifier)
C	EIP-1967: Proxy Logic Storage Slot	0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc
D	EIP-1967: Proxy Beacon Storage Slot	0xa3f0ad74e5423aebfd80d3ef4346578335a9a72aeaee59ff6cb3582b35133d50
E	EIP-1822: Universal Upgradeable Proxy Standard	0xc5f16f0fcc639fa48a6947836d9850f504798523bf8c9a3a87d5876cf622bcf7
F	EIP-2535 : Diamonds Proxy	interface "IDiamondCut"
G	OpenZeppelin Proxy	0x7050c9e0f4ca769c69bd3a8ef740bc37934f8e2c036e5a723fd8ee048ed3f8c3

ple storage slot analysis by first determining whether the delegatecall is used and how the target address is handled. We then trace whether the implementation address is hardcoded or stored in a modifiable slot, ensuring a more accurate classification of proxies. While other detection methods, such as those by Bodell et al. (Bodell III et al., 2023), Ebrahimi et al. (Ebrahimi, Adams, Oliva, & Hassan, 2024), and Salehi et al. (Salehi et al., 2022), utilize source code or bytecode, these approaches may exclude unverified contracts or those with no upgrading requests in their transaction history. Our bytecode-based method provides a more comprehensive solution by identifying all potential upgradeable proxies, regardless of their verification status or transaction activity. Figure 1 presents a flowchart of the process for identifying different upgradeable proxy contracts. If a proxy contract does not align with the identified types, we categorize it as *Other Upgradeable Proxy Contract* to ensure comprehensive classification within our analysis, as shown in Figure 3. Additionally, Table 2 lists the distinct storage slots or identifiers associated with each proxy type and their label in the flow chart.

After identifying upgradeable proxy contracts, we statistically analyze how prevalent upgradeable proxy contracts are and what is the most commonly used upgrading pattern. By the end of this step, we have a list of upgradeable contracts, their address, and upgrading patterns.

3.4 Historical Versions of Smart Contracts

There is no guarantee that all upgradeable contracts will be modified and upgraded. RQ2 aims to analyze how likely an upgradeable contract is to be upgraded. Hence, we trace historical versions of each upgradeable smart contract, identified in RQ1, to answer this research question. We trace versions by analyzing the logs and events of the upgradeable contracts. The logs and historical blocks for each upgradeable contract are obtained from the Ethereum ETL dataset. We identify the upgrade request (if available) from these logs and save the new version address and details. For instance, some upgradeable contracts emit events with new contract addresses whenever there is a new

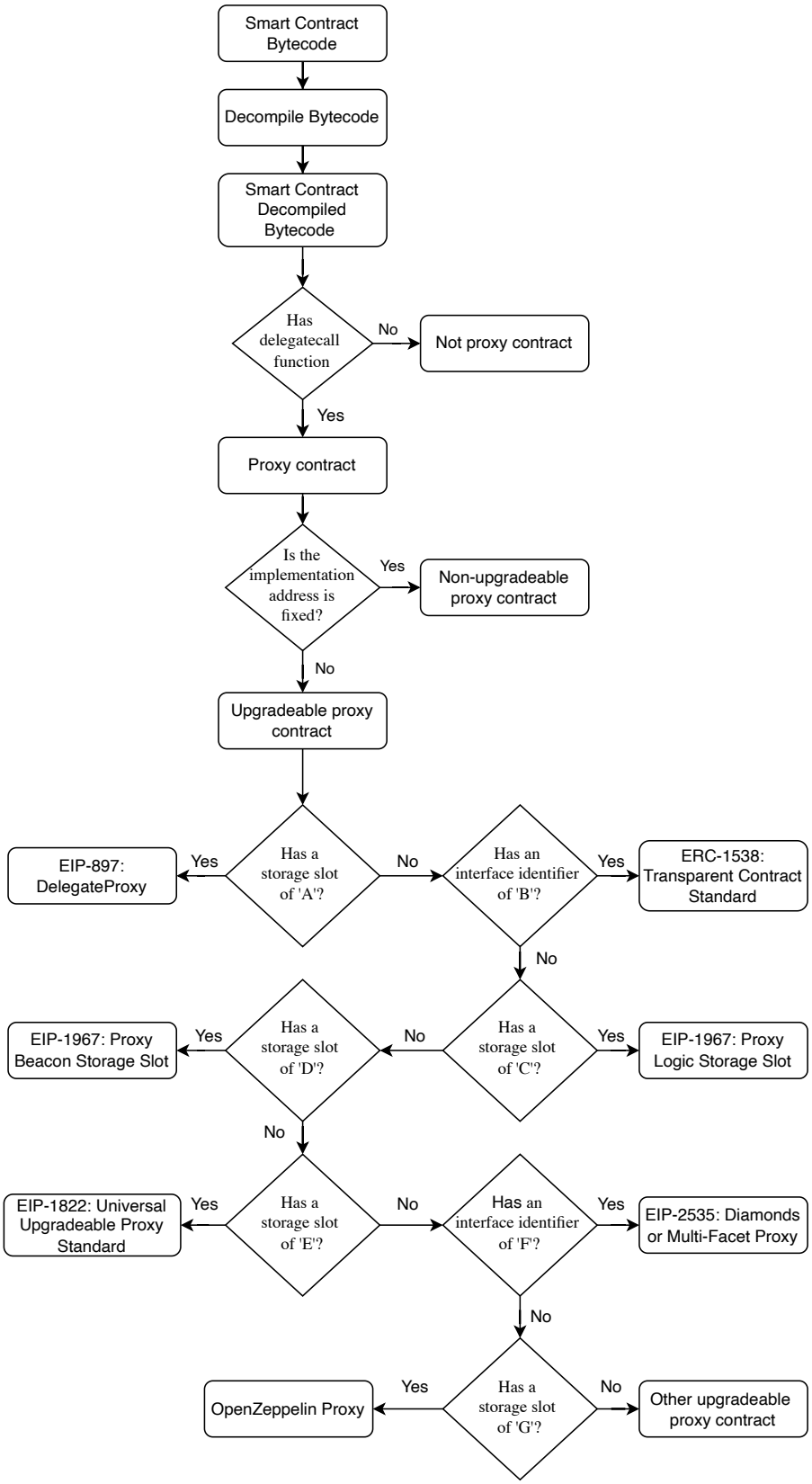

Fig. 3 Flow chart to distinguish different upgradeable proxy contracts

Fig. 4 Sample of emitted upgrade contract events

upgrade for the smart contract. In these cases, we trace all the emitted events for the smart contract and extract the old and new version addresses. Figure 4 shows a sample of emitted events when a contract is upgraded obtained from Etherscan.

Tracing the historical versions of smart contracts poses significant challenges, particularly in the context of event tracing. It is essential to acknowledge that upgrading event signatures vary and are influenced by the specific upgrade strategy employed. Moreover, not all contracts adhere to standard names for upgrading events. These variations in event naming conventions and signatures further complicate the tracing process. For example, a slight alteration in the upgrade event signature can lead to entirely different event hashes, rendering event tracing susceptible to discrepancies.

To address these challenges, we conducted a preliminary study on the existing event naming conventions, including those provided in the standards. For this purpose, we analyzed a sample of 5,000 upgradeable proxy contracts identified from the dataset in RQ1.

To ensure that our sample adequately represents the broader dataset, we employed a stratified random sampling strategy across different deployment years and proxy types. The dataset was stratified based on two key attributes: the deployment year and the proxy type. Contracts were randomly selected within each stratum in proportion to their frequency in the full dataset. For example, if a particular combination (e.g., 'EIP-1967: Proxy Logic Storage Slots' deployed in 2021) represented 10% of the total dataset, approximately 500 contracts of this type and year were included in the sample. This approach ensured that the sample accurately reflects the diversity and distribution of the dataset.

To validate the representativeness of our sample, we performed a Chi-square test for goodness of fit, comparing the observed frequencies of unique proxy types and year combinations in our sample with their expected frequencies in the full dataset. The test is based on the following formula:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

where O_i represents the observed frequency of a particular combination of 'Proxytype' and 'Year' in the sample, and E_i is the expected frequency based on the full dataset distribution. The calculated Chi-square statistic was 1.912, with a p-value of 1.0, indicating no statistically significant difference between the sample and the full dataset distribution (significance level: $p < 0.05$). This result supports the reliability of our sample for subsequent analyses.

We manually analyzed these 5,000 proxy contracts to identify events that might indicate an upgrade. We prioritized events that included an address as an attribute, as these often indicate changes to the contract's logic or version address. Initially, we identified 15 potential upgrade-related events based on the emitted logs. To ensure the relevance of these events, we further examined the source code of the contracts to determine when these events are typically emitted and whether they genuinely relate to upgrade actions. Through this detailed analysis, we excluded six events deemed irrelevant for tracing upgrades, as they were primarily associated with unrelated activities. The remaining nine events considered relevant for this study are:

- Upgraded(address)
- NewImplementation(address, address)
- ProxyUpdated(address, address)
- FunctionUpdate(bytes4, address, address, string)
- Upgraded(uint256, address)
- TargetUpdated(address)
- ImplementationUpdated(address)
- NewImplementation(bytes32, bytes32, address)
- ImplChanged(address, address)

These identified events facilitated tracing smart contract versions within proxy contracts and mapping them to the identified upgradeable proxies in RQ1, resulting in a comprehensive dataset of upgradeable proxy contracts, their corresponding upgraded versions, and the upgrades they have undergone. It is important to note that from the standards discussed in Section 3.3, only three explicitly recommended the use of events to signal upgrades: ERC-1538: Transparent Contract Standard, EIP-1967: Proxy Storage Slots, and EIP-2535: Diamonds or Multi-Facet Proxy. Other standards, such as EIP-1822: Universal Upgradeable Proxy Standard (UUPS) and EIP-897: DelegateProxy, do not include event emissions as part of their upgradeability specifications. This difference affects our event-based detection method, as proxies following these standards may not emit any upgrade events, making it more challenging to track upgrades for such contracts.

Limitations of Event-Based Detection: While our primary method for detecting upgrades relies on emitted events, we acknowledge that this approach has limitations. Not all smart contract upgrades emit relevant events, and identifying upgrades solely through events may miss instances where no such

events are logged. In fact, our analysis showed that out of 1.3 million upgradeable proxy contracts, approximately 12.57% did not emit any upgrade-related events. We explored alternative methods, such as tracing function calls (e.g., `upgradeTo` or similar functions). However, these approaches present several challenges: the same ambiguity faced with event signatures is also applicable here, where function signatures in Ethereum smart contracts are not standardized and can vary significantly across different implementations, making it difficult to identify upgrade-related actions without precise knowledge of the function signatures involved. Furthermore, unlike event parameters, function inputs are stored as part of the transaction data. They are not indexed, necessitating an iterative process over all relevant transactions to manually decode each input, which is computationally intensive given the dataset's size. Analyzing and decoding transaction data for potentially millions of transactions introduces significant computational overhead. Even if upgrade-related function calls are identified, verifying them by monitoring storage changes adds further complexity and uncertainty. Given these challenges, our reliance on event-based detection remains the most feasible and practical method for identifying smart contract upgrades in this context. While this method may not capture every upgrade, especially those not emitting relevant events, it reasonably balances feasibility and accuracy. This limitation is further discussed in the Threats to Validity section.

3.5 Analyzing Post-Upgrade Changes in Smart Contracts

The objectives of RQ3 are to identify and analyze the specific modifications undertaken in smart contract upgrades. This study explores how traditional software maintenance classifications, such as corrective and perfective, apply to smart contracts. In this context, we aim to ascertain and analyze the reflection of these established maintenance activities within the domain of smart contracts.

For RQ3, we categorize post-upgrade changes into three types: fixing vulnerabilities, feature modification, or optimizing gas cost, based on common motivations for smart contract upgrades observed in both academic literature (Bodell III et al., 2023; J. Chen et al., 2021; Salehi et al., 2022) and industry practice.[1,2,3] These categories align with established software maintenance classifications, where fixing vulnerabilities corresponds to corrective maintenance, while feature modification and gas optimization align with perfective maintenance.

We focused on these three categories for several reasons:

- Security concerns, feature updates, and gas efficiency are among the most pressing issues in smart contract development (J. Chen et al., 2021; T. Chen, Li, Luo, & Zhang, 2017; Atzei, Bartoletti, & Cimoli, 2017). Industry resources from major platforms such as Ethereum[2] and OpenZeppelin.[1,3] In practice, security patches prevent vulnerabilities, feature modifications enhance contract functionality, and gas optimizations improve cost efficiency.

– These categories are observable and can be automatically detected through code analysis or using tools without requiring extensive manual effort.
– These factors are crucial for both users and developers, as they directly affect the usability and cost-effectiveness of the contract (H. Chen, Pendleton, Njilla, & Xu, 2020; Soud, Liebel, & Hamdaqa, 2024; T. Chen et al., 2017; Albert, Correas, Gordillo, Román-Díez, & Rubio, 2020; Nelaturu, Beillahi, Long, & Veneris, 2021).

Identifying adaptive maintenance (updates necessitated by environmental shifts) and preventive maintenance (mitigating future issues) is challenging due to inherent difficulties in their application and identification (J. Chen et al., 2021). Therefore, modifications that might fall under these categories or do not align with the above three post-upgrade changes are categorized under an "other" classification.

Our categorization is based on observable changes in the smart contract code rather than inferring the root causes or developer intentions behind these changes. Our approach does not aim to identify the developer's intention, as this cannot be reliably determined from the code alone. Establishing the true purpose behind a change would require extensive manual investigation, including examining developer communications, documentation, and possibly interviewing developers, which is impractical given the scale of our dataset and resource constraints. Therefore, we focus on observable outcomes to provide a practical categorization and quantification of changes without speculating on the underlying motivations. Multiple types of changes may coexist within a single upgrade. For example, if developers delete lines of code to fix a vulnerability, this action could also reduce gas costs or modify the contract's functionality. In such cases, even though the primary change may be to fix a vulnerability, the observable outcomes could fall into multiple categories in our study.

In this step, we use the smart contract versions from RQ2 to to identify the types of changes in each upgrade. Our analysis focuses exclusively on smart contracts with verified source code. This is because certain categories, such as fixing vulnerabilities and feature modifications, require a detailed examination of the source code to assess the nature of the changes accurately. Analyzing contracts without source code would limit our ability to apply the full categorization method, potentially introducing biases. To identify the changes in post-upgrades, it is necessary to find the code changes in each smart contract version (upgrade). For code change detection between the identified smart contract versions (from RQ2), we use Git diff,[15] a popular and powerful tool that is well-suited for comparing different code versions. The Git-diff tool results help us locate the code changes for each upgrade. To analyze security issues

[1] `https://docs.openzeppelin.com/learn/upgrading-smart-contracts`
[2] `https://ethereum.org/en/developers/docs/smart-contracts/upgrading/`
[3] `https://blog.openzeppelin.com/the-state-of-smart-contract-upgrades`
[15] `https://git-scm.com/docs/git-diff`

in the old and new versions, we use the SmartBugs framework[16] to detect vulnerabilities in smart contract versions. SmartBugs integrates several well-established security analysis tools, such as Solhint[17], Mythril[18], Slither[19], and Semgrep[20], which are continuously maintained and updated to address new vulnerabilities. These tools are also used by industry leaders like OpenZeppelin, underscoring their reliability and relevance in real-world applications. The framework provides a unified platform that allows these tools to run in parallel, reducing configuration time and simplifying the analysis process.

Furthermore, we analyze whether gas was optimized in the new versions of the contract. In this step, we estimate and compare the deployment gas cost of the contract versions by conducting an on-chain analysis with historical data from Etherscan. Specifically, we focus on the actual gas used during contract deployment by observing real-world transactions from verified contracts. This method captures the gas consumption data as recorded on the Ethereum blockchain. Deployment gas cost serves as a proxy for optimizations at the contract level, including changes in contract size, storage initialization, and overall complexity. While it does not account for the gas costs of individual function executions, it provides a reliable overview of structural optimizations in newer versions of the contracts.

For minimal changes, such as a single line of code that does not significantly modify a function, the change is placed in the "other" category. For instance, changing a value (e.g., adjusting a constant fee from 2% to 3%) without impacting security or performance is classified as "other" because it does not align with vulnerability fixes, feature modifications, or gas optimizations. This category allows us to account for minor adjustments or policy changes that do not fit the core categories in this study.

To systematically categorize these changes, we employed the following approach:

1. Locate changes between smart contract versions using the Git diff tool.
2. Run the SmartBugs suite to execute multiple security analysis tools in parallel on both versions of the smart contract to detect vulnerabilities. If there is a security vulnerability in the first version that was not found in the second version, mark it as a bug fix. (this aligns with corrective maintenance).
3. Identify any changes that were made between the two versions that do not address a security vulnerability. If the second version added lines of code that do not fix a bug, mark it as a new feature (perfective maintenance). Otherwise, if lines or functions were removed without fixing any vulnerabilities in the code, label it as a deleted feature.

[16] `https://hub.docker.com/u/smartbugs`
[17] `https://github.com/protofire/solhint`
[18] `https://github.com/Consensys/mythril`
[19] `https://github.com/crytic/slither`
[20] `https://github.com/semgrep/semgrep`

4. Compare the deployment gas costs of both versions of the smart contract using historical transaction data from Etherscan. If the gas cost has decreased in the second version, mark it as a gas optimization (perfective maintenance).
5. For any modifications that do not fall into the categories of bug fix, new feature, deleted feature, or gas optimization, and for minimal changes(e.g., small adjustments to constant values or parameters), label these as "other." These changes might align with adaptive or preventive maintenance, depending on whether they aim to prevent future issues or adjust to changes in the environment.

Algorithm 1 demonstrates the labeling process of post-upgrade changes based on the above steps.

3.6 Upgradeability Impact on Contract's Activity level

To analyze the impact of upgrading the contract on its usage (RQ4), we use the received transaction numbers for each version to get insight into the activity level. The transaction number is exported from the data collected from Etherscan. We compared the activity levels between each consecutive version to identify changes in transaction patterns after upgrades. In this analysis, we focused exclusively on upgradeable proxy contracts with multiple versions. Non-active upgradeable proxies, which have a single version were excluded to avoid introducing bias. This ensures that only proxies actively contributing to the relationship between upgrades and activity levels were included in the analysis.

Since smart contract versions can have different lifespans, it is necessary to analyze the contract's activity level while considering its lifespan. The contract version lifespan is measured from when the contract was deployed until a new version was created. The age or lifespan is measured using the following equation:

$$\textit{lifespan}(V_a) = DT(V_{a+1}) - DT(V_a) \tag{1}$$

where V_a is the target version for which we want to calculate the lifespan $\textit{lifespan}(V_a)$, V_{a+1} is the next version, and $DT(V)$ is the deployment time of version V. If V_a is the latest version, we consider the collection date instead of $DT(V_{a+1})$ as the end of the deployment period. To answer RQ4, we use a regression model with transaction numbers as the dependent variable and version lifespan as an independent variable. By including version lifespan in the model, we control for any differences in activity level due to the different lifespan of smart contract versions. Initially, we consider a linear regression model to examine if a linear relationship exists between the transaction count and the version lifespans. The equation can represent the linear regression model:

$$\text{Transaction numbers} = \beta_0 + \beta_1 \times \text{Version lifespan} + \epsilon \tag{2}$$

Algorithm 1 Identifying Post-Upgrade Changes in Smart Contract

Input: $version1$, $version2$
Output: $post\_upgrade\_changes$
$versions\_diff \leftarrow git\_diff(version1, version2)$
$vulnerabilities1 \leftarrow smartbugs(version1)$
$vulnerabilities2 \leftarrow smartbugs(version2)$
$bug\_fixes \leftarrow []$
$new\_features \leftarrow []$
$deleted\_features \leftarrow []$
$gas\_optimizations \leftarrow []$
$other \leftarrow []$
for $vulnerability$ in $vulnerabilities1$ **do**
  if $vulnerability \notin vulnerabilities2$ **then**
    $bug\_fixes.append(vulnerability)$
  end if
end for
for $line$ in $versions\_diff.added\_lines$ **do**
  if $line \notin vulnerabilities1$ **and** $line \notin vulnerabilities2$ **then**
    $new\_features.append(line)$
  end if
end for
for $line$ in $versions\_diff.removed\_lines$ **do**
  if $line \notin vulnerabilities1$ **and** $line \notin vulnerabilities2$ **then**
    $deleted\_features.append(line)$
  end if
end for
$gas\_cost1 \leftarrow calculate\_gas\_cost(version1)$
$gas\_cost2 \leftarrow calculate\_gas\_cost(version2)$
if $gas\_cost2 < gas\_cost1$ **then**
  $gas\_optimizations.append("Gascostdecreased")$
end if
if len$(bug_fixes) > 0$ **then**
  $post\_upgrade\_changes.append("Bugfix")$
end if
if len$(new_features) > 0$ **then**
  $post\_upgrade\_changes.append("Newfeature")$
end if
if len$(deleted_features) > 0$ **then**
  $post\_upgrade\_changes.append("Deletedfeature")$
end if
if len$(gas_optimizations) > 0$ **then**
  $post\_upgrade\_changes.append("Gasoptimization")$
end if
if len$(post_upgrade_changes) == 0$ **then**
  $post\_upgrade\_changes.append("Other")$
end if
return $post\_upgrade\_changes$

where β_0 is the intercept, β_1 is the coefficient for the independent variable representing the Version lifespan, and ϵ represents the error term of the model.

In our analysis, recognizing that a linear relationship might not adequately capture the complexities of our data, we have considered the application of random forest regression, a more complex machine learning model that uses an ensemble of decision trees to make predictions. We conduct an exploration of

random forest regression to assess its effectiveness in our context. This technique allows us to capture non-linear relationships and interactions between variables that a simple linear model may overlook. The random forest model does not have a simple equation like linear regression, as it is based on multiple decision trees that collectively contribute to the final prediction. To assess the performance of our models (linear and random forest), we use the Mean Squared Error (MSE) and the R-squared (R^2) score. The MSE is calculated as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{3}$$

where y_i is the observed value, $\hat{y}_i$ is the predicted value by the model, and n is the total number of observations in the dataset. The R-squared score, represented as:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{4}$$

where $\bar{y}$ is the mean of the observed values. The R^2 score measures the proportion of the variance in the dependent variable that is predictable from the independent variable(s). An R^2 score closer to 1 indicates that the model explains a large portion of the variance in the dependent variable.

Based on the calculated activity levels, we analyze the impact of upgrading on the activity levels of different versions of the contract.

4 Results

This section presents the results of our empirical study, addressing the research questions and outlining our key findings. The project source codes and the generated dataset are openly available in the project repository.[21]

4.1 RQ1: Prevalence of Upgradeable Smart Contracts

Figure 5 presents a dual donut chart delineating the distribution of smart contract types in a dataset of 44 million contracts, focusing on upgradeable proxy contracts. The smaller donut chart categorizes these contracts into three distinct groups: Non-Proxy Contracts, Non-Upgradeable Proxy Contracts, and Upgradeable Proxy Contracts. The larger donut chart focuses on the Upgradeable Proxy Contracts segment, which provides insight into the distribution of different upgradeable standards mentioned in Section 3.3.

[21] https://drive.google.com/drive/folders/1xlY7suki5TqiOF7mipKspfBbTb888-iz?usp=sharing

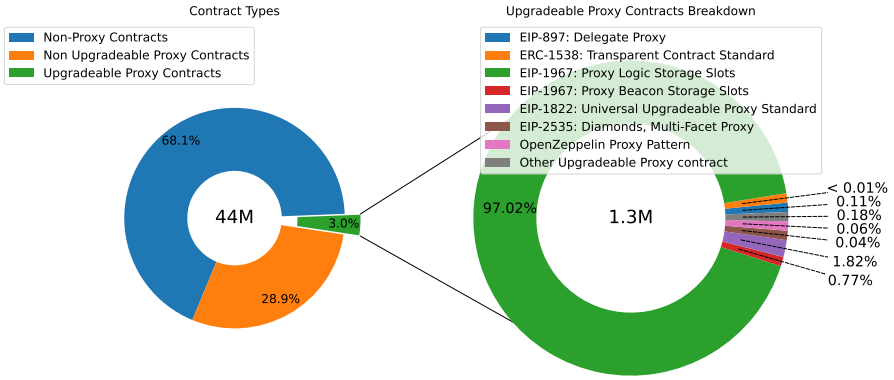

Fig. 5 RQ1:Prevalence of upgradeable smart contracts

Observation RQ1.1: Upgradeable Proxy Contracts comprise only a small percentage, specifically 3%, of the total dataset. The largest segment, depicted in blue, represents non-proxy contracts, signifying their prevalence as the majority within the dataset. The orange segment, slightly smaller in size, corresponds to non-upgradeable proxy contracts. The smallest segment, depicted in green, is of key interest to our analysis as it represents upgradeable proxy contracts. The results from the chart indicate that Upgradeable Proxy Contracts account for only a small fraction of the total smart contracts quantified as 3% of the dataset. This observation points to a limited adoption of upgradeable patterns in smart contracts. Despite their availability and potential advantages, the relatively small share of upgradeable contracts suggests that they are not the primary choice in the smart contract domain. Factors influencing this include the complexity of implementing and managing upgradeable contracts or that developers are favoring the stability and predictability of non-upgradeable contracts over the potential benefits of upgradeability.

Observation RQ1.2: Among upgradeable contracts, *EIP-1967: Proxy Logic Storage Slots* is the most utilized standard, suggesting a favored approach within the subset of upgradeable contracts. Focusing on the Upgradeable Proxy Contracts segment, a secondary donut chart provides insight into the distribution of different upgradeable standards. Among these, the EIP-1967: Proxy Logic Storage Slots emerges as the most utilized, followed by other standards like the EIP-1822 Universal Upgradeable Proxy Standard and EIP-1967 Proxy Beacon Storage Slots. This distribution within the upgradeable contracts highlights a preference for certain standards, underscoring the EIP-1967 Proxy Logic Storage Slots as the principal choice.

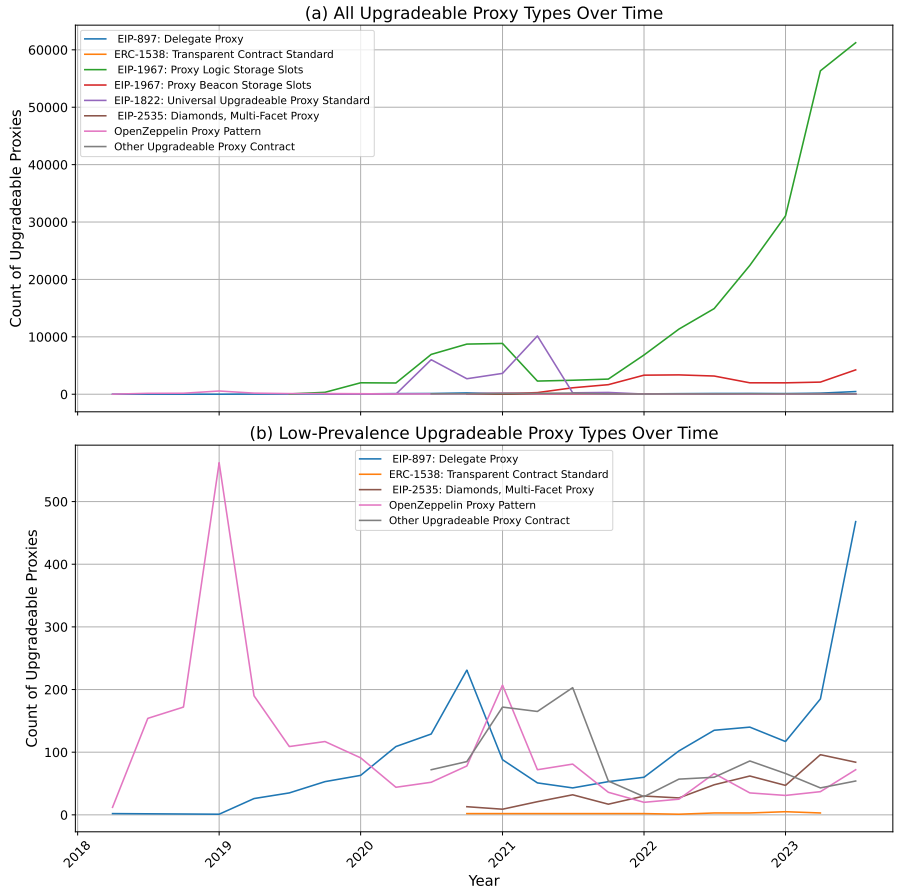

Fig. 6 Time series analysis of the adoption of upgradeable proxy types

Observation RQ1.3: Different proxies exhibit varying trends over time, highlighting dynamic shifts in technology adoption and standard preferences in the smart contract domain. Figure 6 presents a detailed analysis of proxy usage over time, separated into two subfigures for clarity. Figure 6 (a) shows the counts of all upgradeable proxy types from 2018 to mid-2023. Figure 6 (b) focuses specifically on the low-prevalence upgradeable proxy over the same period. In both subfigures, the x-axis represents time, divided into quarters, to provide a temporal view of the data. The y-axis quantifies the count of upgradeable proxies, giving insight into the prevalence and adoption of different upgradeable proxy types over time. Each line corresponds to a different proxy type, their trajectories reflecting the evolving count of each across consecutive quarters. Notably, the chart encapsulates a spectrum of trends: while some proxy types exhibit a steady presence, others reveal dynamic patterns, including phases of decline, stability, or growth. An adjustment was made to account for partial data coverage in the last quarter

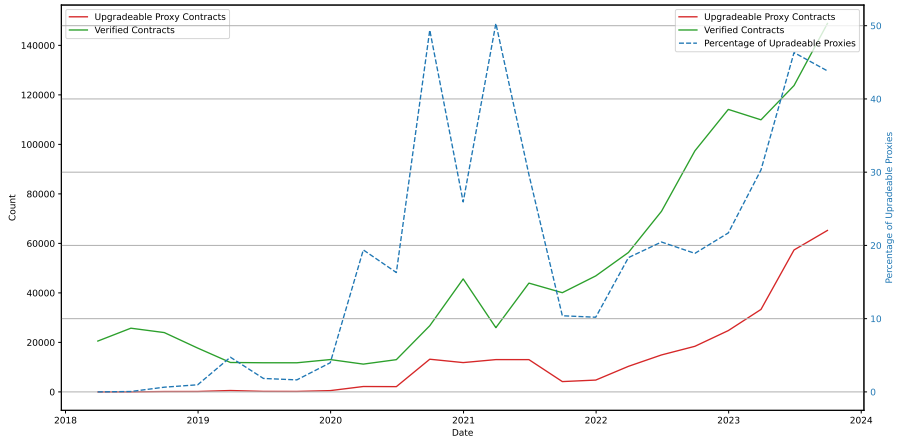

Fig. 7 Trends in upgradeable proxy versus total verified smart contract deployments

represented in the data. Typically, each quarter comprises 90 days; however, the last quarter included only 15 days of data. To mitigate the potential distortion this discrepancy could introduce, the count for the last quarter was scaled to reflect a full quarter's worth of data. This scaling involved multiplying the observed count by a factor of 6 (i.e., 90/15), a methodological decision to preserve the integrity of the trend analysis.

Delving into specific instances, the *Proxy Logic Storage Slots (EIP-1967)* line presents an intriguing narrative. Initially, it shows a substantial presence, followed by a period of decline. This dip could indicate shifts in technology adoption, perhaps due to the emergence of more efficient or versatile proxy solutions. Concurrently, there is an uptick in the adoption of *UUPS (EIP-1822)*, suggesting a possible correlation. This period also coincides with the introduction of the *Diamond Proxy (EIP-2535)*, which may have influenced the usage dynamics of other proxies. However, it is worth noting that the decline in the usage of *EIP-1967* was temporary, as there was a subsequent increase in its adoption. Interestingly, during this phase, *EIP-1822* experienced a decrease in its usage, which could be due to the simplicity and efficiency of *EIP-1967*, which might have regained favor over *EIP-1822*.

In Figure 6 (b), some proxies, such as The *Diamond Proxy (EIP-2535)* trend, remain relatively stable throughout the observed period. This stability indicates a consistent, albeit niche, adoption. In contrast, the *Delegate Proxy (EIP-897)*, which started with low adoption, shows a notable increase in 2023, suggesting a resurgence in interest after years of stagnation. The OpenZeppelin Proxy Pattern exhibits a more dynamic trend, with a sharp peak in 2019 followed by a decline and eventual stabilization at a lower level.

Observation RQ1.4: There is steady growth in the deployment of upgradeable smart contracts over time. Figure 7 presents the trends in upgradeable proxy contract deployments compared to the overall number of

verified smart contracts over time. The x-axis outlines the timeline by quarters, illustrating the progression over several years. The primary y-axis on the left, colored in black, represents the count of contracts, tracking both upgradeable proxy contracts and total verified contracts. As in Figure 6, the last quarter's data was scaled due to incomplete data. A secondary y-axis on the right, highlighted in blue, details the percentage of upgradeable proxy contracts relative to the total verified contracts, offering insights into the adoption rate of upgradeable proxies within the broader smart contract ecosystem. In Figure 7, two distinct lines trace the evolution of contract counts: the red line for upgradeable proxy contracts and the green line for total verified contracts. These lines encapsulate the quantitative growth in each category, revealing patterns of increase, stability, or fluctuation over the observed period. Additionally, a blue dashed line marks the percentage of upgradeable proxy contracts, providing a relative measure of their prevalence against the backdrop of total contract activities.

In early 2018, when upgradeable proxy contracts were first introduced, there was no recorded use of them, highlighting their initial absence in the smart contract landscape. This period marked the beginning of an exploration into more flexible smart contract mechanisms beyond the traditional immutable contracts.

Following their introduction, the percentage of deployed upgradeable proxy contracts began to increase, indicating a growing interest and recognition of their potential benefits. This uptick in adoption showcased the development community's initial steps towards embracing the flexibility that upgradeable proxies offered for smart contract development. The trend toward adopting upgradeable proxy contracts consistently rose from 2019 through the end of 2021. Despite experiencing fluctuations during this period, which likely reflected the community's ongoing experimentation and assessment of upgradeable proxies, the overall direction was clear: a steady move towards greater adoption. These fluctuations were part of the natural process of integrating new technology into existing practices as developers weighed the advantages of upgradeability against the foundational principles of blockchain technology.

Starting in 2022, the adoption of upgradeable proxy contracts witnessed a significant increase, which is supported by the Year-over-Year (YoY) growth rate. The YoY growth rate is calculated as follows:

$$\text{YoY} = \left(\frac{A - B}{B} \right) \times 100$$

Where**A** is the number of upgradeable contracts in the current year and **B** is the number of upgradeable contracts in the previous year.

For 2022, the YoY growth in upgradeable proxy contracts was approximately 95.4%, compared to 19.4% in 2021. This marked acceleration highlights the significant growth in the use of upgradeable proxies starting in 2022.

The growing adoption reflects the community's evolving understanding of the importance of upgradeability, particularly as smart contracts need to remain adaptable to technological, regulatory, and operational changes. Several

well-known projects demonstrate this need for adaptability. For example, Aave upgraded its borrowing rate mechanism to improve flexibility in response to the evolving DeFi landscape [22]. Similarly, Compound's Compound III release focused on optimizing gas efficiency and improving risk management, driven by both regulatory and user needs [23].

Observation RQ1.5: Despite representing only 3% of the total dataset, upgradeable proxy contracts play a crucial role in several high-value and widely used protocols in the Ethereum ecosystem. To highlight the importance of these contracts, we identified several major projects, as shown in Table 3 that rely on upgradeable proxies, including Compound, OpenSea, Aave, Synthetix, and Harvest Finance. These projects span various domains, from decentralized finance (DeFi) to non-fungible tokens (NFTs), where upgradeability is essential for maintaining adaptability and responding to changing requirements.

Table 3 Major Projects Using Upgradeable Proxies

Project	Domain	Proxy Address	TVL
Compound[24]	DeFi	0x3d9819210a31b4961b30ef54be2aed79b9c9cd3b	$2.022 billion
OpenSea[25]	NFTs	0x7Be8076f4EA4A4AD08075C2508e481d6C946D12b	-
Aave[26]	DeFi	0x317625234562B1526Ea2FaC4030Ea499C5291de4	$1.047 billion
Synthetix[27]	DeFi	0x23348160D7f5aca21195dF2b70f28Fce2B0be9fC	$279.43 million
Harvest Finance[28]	Yield Aggregators	0x1851a8fa2ca4d8fb8b5c56eac1813fd890998efc	$19.48 million

Collectively, these projects account for over $3.3 billion in Total Value Locked (TVL). Although it is not possible to generalize that all upgradeable proxies are highly significant, these projects demonstrate that upgradeable proxies are essential for major, impactful projects. This reinforces our argument that despite their low overall percentage, upgradeable proxies control substantial economic activity, underscoring their importance in the ecosystem.

RQ1. How prevalent are upgrading patterns in smart contracts?

Answer: The results show that Upgradeable Proxy Contracts constitute a mere 3% of the dataset, suggesting a limited prevalence of upgradeable patterns in the smart contract landscape. This restrained adoption may stem from perceived complexities in implementing and managing such contracts or a preference for the stability of non-upgradeable contracts. Despite this, the distribution of upgradeable proxy standards reveals a preference for EIP-1967, indicating nuanced selections within the upgradeable subset. Further temporal analysis shows varying trends among proxy types, with some experiencing phases of decline and others gaining

[22] https://medium.com/aave/aave-borrowing-rates-upgraded-f6c8b27973a7

[23] https://medium.com/compound-finance/compound-iii-is-live-a7983dee7e60

adoption, particularly noting the dynamic interplay between EIP-1967 and EIP-1822 standards. This reflects shifting preferences and technological advancements in the smart contract domain. Moreover, a consistent increase in the deployment of upgradeable contracts from 2020 to 2021 signifies a growing developer inclination towards flexible and updateable contract structures. Despite a slight deceleration post-2021, the trend towards upgradeability remains upward. Thus, while upgradeable proxy contracts initially represented a small fraction of the dataset, the evolving adoption patterns and preference shifts among developers suggest a gradual but discernible change in the landscape of smart contract development.

4.2 RQ2: Historical Versions of Smart Contracts

To address RQ2, we focused on the number of smart contracts that were upgraded at least two times, based on the events discussed in Section 3.4.

Observation RQ2.1: Only a small fraction of upgradeable contracts have been upgraded. From the dataset of 1.3 million upgradeable proxy contracts, a relatively small fraction, specifically 4,397, have been upgraded, resulting in 14,990 distinct smart contract versions. This observation underscores a general tendency within the smart contract community: despite the technical feasibility of frequent upgrades, the actual incidence of such upgrades is comparatively low.

Observation RQ2.2: The *Upgraded(address)* event is the most common, while others are rare, indicating different levels of standardization and application in upgrade processes.

The prominence of the *Upgraded(address)* event is primarily due to its inclusion in the EIP-1967 standard, which explicitly recommends emitting this event during upgrades. As highlighted in RQ1, the EIP-1967 standard is the most widely adopted approach for upgradeable proxies, which explains the widespread use of this event. Table 4 provides an overview of the frequency of upgrade events. In contrast, events such as *ImplChanged(address,address)* are rare, as they are not part of any standardized documentation.

This observation is particularly important for developers, auditors, and platform operators. Developers benefit from the standardization of events like *Upgraded(address)* as it ensures compatibility with monitoring tools and simplifies upgrade tracking. For auditors, this consistency aids in validating upgrades and assessing contract changes. Platform operators can monitor these events to maintain operational stability and identify potential risks associated with contract upgrades.

Observation RQ2.3: A significant number of contracts undergo minimal upgrades, with the most common version number being 2. Figure

Table 4 Frequency of different upgrade events in smart contracts

Event Type	Number of Upgradeable Proxies
Upgraded(address)	3,715
NewImplementation(address,address)	571
ProxyUpdated(address,address)	31
FunctionUpdate(bytes4,address,address,string)	21
Upgraded(uint256,address)	20
TargetUpdated(address)	19
ImplementationUpdated(address)	9
NewImplementation(bytes32,bytes32,address)	7
ImplChanged(address,address)	4

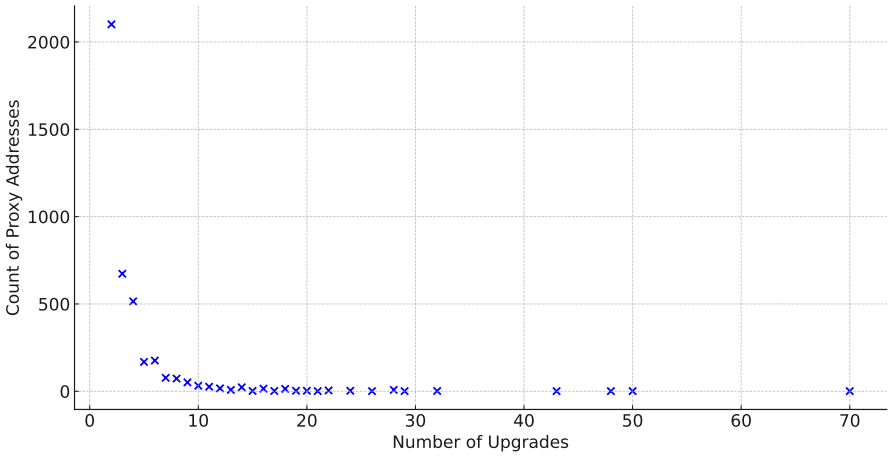

Fig. 8 Historical smart contract versions (RQ2)

8 illustrates the distribution of versions across upgradeable smart contracts, revealing key insights into their upgrade patterns. The x-axis of the figure shows the number of versions, while the y-axis represents frequency. Each cross indicates the number of smart contracts with exactly that number of versions. The figure shows that the most frequent version number across these contracts is 2, highlighting a notable inclination towards at least a single upgrade cycle among these contracts. In contrast, the dataset reveals that the highest recorded version number is 70. Although such high version numbers are less common, as indicated by the decreasing density of crosses in the higher version range of Figure 8, they point to some contracts being subject to extensive upgrading, undergoing numerous iterations over their lifecycle. Moreover, the average number of versions per contract is approximately 3.90, suggesting a general trend where most contracts are subjected to limited upgrades. This pattern is visually affirmed by the pronounced concentration of crosses at lower version numbers, suggesting that most contracts have fewer versions, indicating a trend toward minimal upgrades. As the version number increases, there is a noticeable decline in frequency, highlighting that higher version numbers are increasingly rare.

Table 5 Top five proxies with number of versions and unique implementations

Proxy Address	Number of Versions	Unique Implementations
0x2f6081e3552b1c86ce4479b80062a1dda8ef23e3	70	62
0x62faa8937f71b4896f9b250f675ff89a5f6875cc	50	25
0x1920d646574e097c2c487f69f40814f95d45bf8c	48	23
0x31946680978cefb010e5f5fa8b8134c058cba7dc	43	42
0xb5c9985dc029b37d756938745760747b62ff46f9	32	31

Observation RQ2.4: The dataset reveals a notable replication trend in upgradeable smart contracts, with 50 unique versions being reused multiple times across various contracts. Delving more into the details of the versions used in upgradeable smart contracts, we notice some interesting patterns. Specifically, when examining the dataset for duplicated versions, it is observed that 50 unique versions have been used more than once, either within the same contract (proxy address) or across different contracts. This indicates a practice of reusing specific versions, which could be due to various reasons, such as the popularity of certain contract versions or the reapplication of proven and stable implementations. For instance, the dataset analysis reveals that the most duplicated implementation address is *0xf9e266af4bca5890e2781812cc6a6e89495a79f2*, which has been replicated 394 times across various contracts.

Observation RQ2.5: Analysis of the top 5 proxies indicates two primary upgrade strategies: a majority displaying unique versions for diverse upgrades and a minority favoring version duplication for stability and specific functionalities. The existence of duplicates in smart contract versions draws attention to the need for a more focused analysis, particularly on proxies with a high number of versions. Table 5 demonstrates the top 5 proxies, categorized by the highest number of versions, and examines the uniqueness of these versions.

The table shows interesting patterns about the upgrade strategies of these top five proxies: Proxies *0x62faa8937f71b4896f9b250f675ff89a5f6875cc, 0xb5c9985dc029b37d756938745760747b62ff46f9*, and *0x2f6081e3552b1c86ce4479b80062a1dda8ef23e3* show a high degree of uniqueness in their versions. The close correlation between the number of versions and the number of unique implementations suggests that most versions introduced are distinct, reflecting a strategy of consistent and diverse upgrades. On the other hand, proxies *0x31946680978cefb010e5f5fa8b8134c058cba7dc* and *0x1920d646574e097c2c487f69f40814f95d45bf8c* display a different trend, with a significant portion of their versions being duplicates. This indicates a more conservative approach where certain versions are reused, possibly due to their stability or specific functionality that suits the contract's needs over time.

Observation RQ2.6: The study shows a dominant prevalence of the Proxy Logic Storage Slots (EIP-1967) in upgradeable smart con-

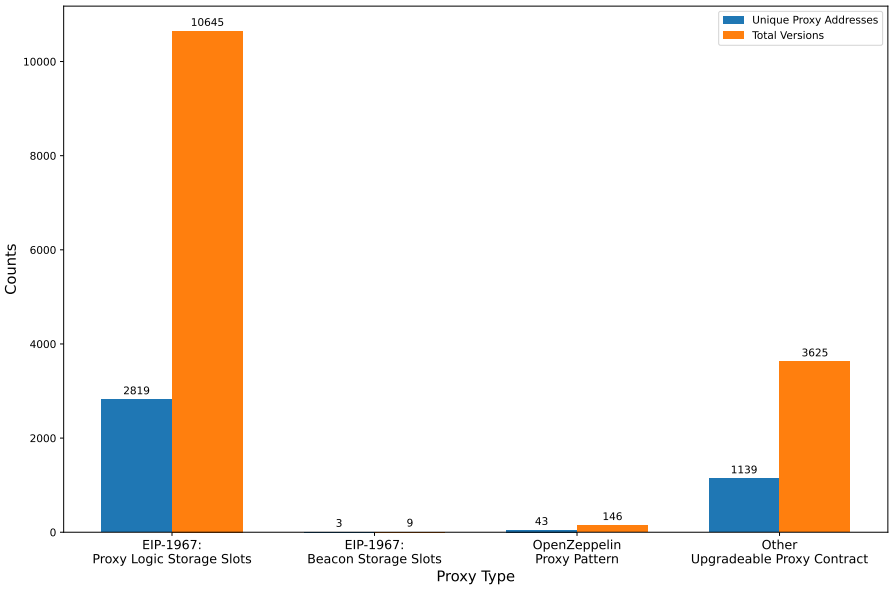

Fig. 9 Smart contract versions versus proxy type

tracts, with less standardized or custom hybrid proxy types also present but with lower adoption rates. Regarding the analysis in the context of RQ1, where we study the prevalence of upgrading approaches, Figure 9 illustrates the distribution of proxy types based on unique proxy addresses and their total versions. This visualization provides a clear comparative view of the proxy types' adoption and number of upgrades. Figure 9 shows the existence of diverse range of less standardized or custom hybrid proxy implementations which is not following any identified standards. We classify these proxy types as a *Other upgradeable proxy contract*. Similar to the results of RQ1, the *Proxy Logic Storage Slots (EIP-1967)* type is the most predominant, both in terms of the number of proxy addresses and total versions. The *OpenZeppelin Proxy Pattern* and *Beacon Storage Slots (EIP:1967)* is reletively low. The missing patterns in Figure 9 do not have any upgrades, indicating that these proxies were not used at all. An interesting observation is the missing upgrades using the approach *UUPS (EIP-1822)* which was the second most used proxy standard in RQ1.

Observation RQ2.7: The trend in smart contract versions shows an increasing trajectory, closely related to the trend in upgradeable proxy contracts.

Figure 10 presents a comparison trend analysis between the deployment of upgradeable proxy contracts and the progression of smart contract versions over time, utilizing a dual-axis approach due to the significant numerical difference between the two datasets. The x-axis represents the timeline by quarters,

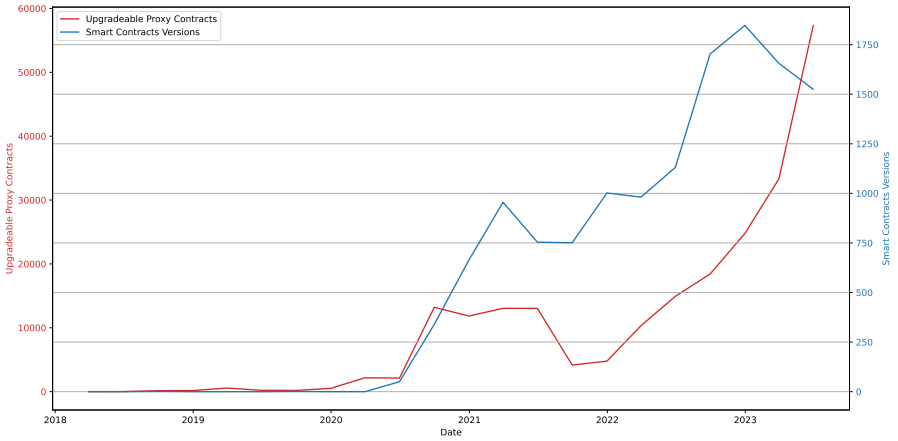

Fig. 10 Comparative trends of upgradeable proxy contracts and smart contract versions

illustrating the progression over several years. The primary y-axis on the left, colored in red, represents the count of upgradeable proxy contracts. A secondary y-axis on the right, highlighted in blue, details the count of smart contract versions. As in Figure 6, the last quarter's data was scaled due to incomplete data.

Initially, the figure illustrates a period where the number of upgradeable proxy contracts is very low, correlating with an absence of smart contract versions. This early stage reflects the foundational need for upgradeable proxies to facilitate contract upgrades, making the initial scarcity of versions expected, given the minimal deployment of upgradeable proxies. As the timeline progresses, both trends begin to exhibit a parallel increase. The rise in upgradeable proxy contracts, after a certain lag period, is followed by an increase in the number of smart contract versions. This sequence underscores the process wherein enhancements and modifications to smart contracts, captured as versions, are inherently linked to the availability and use of upgradeable proxies. The observed lag between these increases is a natural aspect of the development cycle, accounting for the time required to implement, test, and deploy contract upgrades.

A specific observation in the period after mid-2021 shows a temporary decline in the deployment of upgradeable proxy contracts. Interestingly, this does not lead to an immediate decrease in smart contract versions. Instead, an increase in versions is recorded, likely in response to the preceding rise in upgradeable proxies. This demonstrates the delayed effect of proxy deployment on contract versioning, highlighting the time-dependent nature of contract upgrades in response to earlier increases in upgradeable proxies.

The final drop observed in the number of smart contract versions towards the end of the period analyzed can be attributed to a reduction in the number of deployed contracts. This observation might initially suggest a decrease in activity or interest. However, this decrease could reflect a recurring pattern

seen in previous periods, where specific months showed a similar drop in deployment data. Such patterns indicate that the observed decline might not signify a long-term decrease in smart contract deployments. Instead, it could represent cyclical or seasonal variations in contract creation activities, suggesting a normal fluctuation rather than a diminishing trend in the development and deployment of smart contracts.

RQ2. How likely is an upgradeable contract to be upgraded?
Answer: The results show a cautious yet evolving approach towards smart contract upgrades within the community. Despite the inherent technical capabilities for frequent updates, actual upgrade practices are less common, likely due to the complexities and risks associated with post-deployment modifications. The strategies range from dynamic, frequent updates to more conservative, stability-focused versioning, reflecting different contracts' diverse needs and contexts. In essence, the decision to upgrade depends on a balance between technical feasibility, risk management, and specific operational requirements. A temporal analysis shows an increasing trend in both the deployment of upgradeable proxy contracts and the progression of smart contract versions over time. This trend illustrates a natural development cycle where the availability and use of upgradeable proxies facilitate the iterative enhancement of smart contracts, evidenced by the parallel growth in the number of versions.

4.3 RQ3: Analyzing Post-Upgrade Changes in Smart Contracts

RQ3 aims to identify post-upgrade changes in smart contracts and categorize them as fixing vulnerabilities, gas optimization, feature modification, or others.

Figure 11 is a radar chart representing upgrade patterns' distribution in two datasets: All Versions and Smart Contracts Versions with Source Code. It has four axes, each corresponding to a different post-upgrade change category. The axes start from a common central point and extend outward, equally spaced. The values are represented as percentages of the total counts in each category. Each dataset is plotted as a closed loop, forming a shape visually representing each pos-upgrade change's relative emphasis. The All Versions dataset is represented in blue, which includes all smart contract versions identified in RQ2. The Smart Contracts Versions with Source Code dataset is in red, focusing on only smart contract versions with available source codes.

Observation RQ3.1: The availability of source code shifts the focus toward feature modification and Fixing Vulnerability, highlighting the role of accessible code in understanding upgrade causes. The dataset for smart contracts with source code demonstrates a balanced distribution of post-upgrade changes'. Feature modification has 365 instances, representing 31.68% of the dataset, indicating a strong focus on enhancing and evolving contract features. On the other hand, fixing vulnerability presents

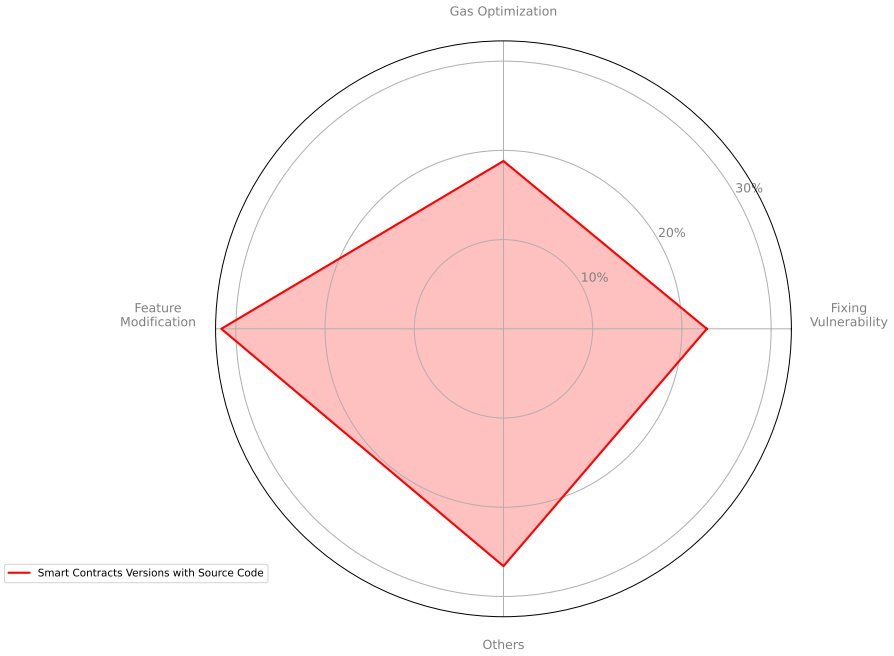

Fig. 11 Post-Upgrade changes in smart contracts (RQ3)

22.83% of the modifications, illustrating a significant commitment to maintaining contract security. Both of these categories rely heavily on the availability of the source code. Hence, it has the same percentage in All versions of the dataset. The feature modification category consists of introducing new features or deleting existing features, where each has the following percentages: 95.07% and 29.59%, respectively. Within a single upgrade, it is possible to have both new features introduced and existing features deleted. This overlap explains why the combined percentages for new and deleted feature modifications exceed 100%. Gas optimization is present in about 18.84% of the dataset, a substantial but relatively smaller proportion compared to all versions. The others category sees a significant reduction to 307 instances, representing 26.65% of this dataset, indicating that the lack of source code contributed to the prevalence of this category.

Observation RQ3.2: Upgrades addressing multiple aspects (security, features, and gas optimization) simultaneously are rare, underlining such upgrades' complexity and resource intensity. We noticed in both datasets the existence of the combinations of these categories, where a single upgrade showed the existence of more than one changes. The pairing of fixing vulnerability with feature modification, particularly introducing new features, stands out, addressing approximately 18.2% of the dataset, where developers are concurrently enhancing security and adding new functionalities.

In the dataset for smart contracts with source code, the co-occurrence of the three specific post-upgrade changes (fixing vulnerability, feature modification, and gas optimization) is rare. These combined modifications are present in 59 instances, constituting approximately 5.12% of the total upgrades. This low percentage highlights the infrequency of such comprehensive upgrades within a single iteration. Concurrently integrating all these modifications (enhancing security, introducing or modifying features, and optimizing for gas efficiency) in one update cycle is a complex and less common practice. This rarity underlines the challenges and potential resource intensiveness of implementing multifaceted upgrades in smart contracts.

RQ3. What Changes Occur in Smart Contracts Post-Upgrade?
Answer: The post-upgrade changes in smart contracts versions, as analyzed through both comprehensive and source code-specific datasets, predominantly focus on diverse motivations (others) in the absence of source code and shift towards a more balanced emphasis on feature modification and security enhancement when source code is available. The rarity of comprehensive upgrades involving simultaneous security, feature, and efficiency improvements highlights the complex and resource-intensive nature of such endeavors in smart contract development.

4.4 RQ4: Upgradeability Impact on Contract's Activity level

As discussed in Section 3.6, we employed a regression model to evaluate the relationship between contract upgrades (versions) and usage (transaction numbers). In this model, transaction numbers were set as the dependent variable, while the version lifespan was the independent variable. This approach aimed to account for variances in activity levels attributable to the different lifespans of smart contract versions.

Observation RQ4.1: A significant portion of versions (90.16%) recorded only one transaction, usually associated with their creation, pointing towards a prevalence of non-active contracts. Interestingly, most contract versions recorded only one transaction (90.16%), typically associated with their creation. This pattern suggested a predominantly non-linear relationship between the number of transactions and the contract versions, casting doubt on the suitability of a linear regression model for this data. Since analyzing the dataset through regression models provides valuable insights into the relationship between contract upgrades, represented by 'Lifespan' and 'Version,' and their usage in terms of 'Total Transactions.'

Observation RQ4.2: The data suggests a non-linear relationship between contract versions and transaction numbers. Initially, we applied a linear regression model to the data. The coefficients obtained were approximately 5.65×10^{-7} for 'Lifespan' and 0.0611 for 'Version'. These coefficients

indicate that an increase in either lifespan or version is associated with an increase in total transactions, with the specified magnitudes. The intercept of the model was around -1.16, marking the point where the regression line would intersect the y-axis.

However, the model's fit to the data, as indicated by the Mean Squared Error (MSE) and the R^2 Score, was less than ideal. The MSE was quite high at 3813.33, and the R^2 Score was -0.03, suggesting that the linear model did not adequately capture the relationship between the variables. The negative R^2 score is particularly telling, as it implies that the model may not be the best fit for this dataset or it fails to account for other influential factors.

Hence, to thoroughly assess these possibilities, we address the first possibility by implementing a more complex model, specifically a Random Forest, to evaluate whether there is any improvement in predictive performance. This approach allows us to explore whether the increased complexity of the model can better capture the underlying patterns in the data and lead to more accurate predictions.

Observation RQ4.3: Both linear and Random Forest (non-linear) models show limitations in capturing the relationship between upgrades and activity, indicating that factors beyond the age and version number influence contract activity. We employed the Random Forest Regressor, a non-linear model for effectively handling complex datasets with multiple variables in response to these limitations. The Random Forest model showed a performance improvement. The MSE reduced to approximately 2378.09, indicating a better fit of the model to the data. Additionally, the R^2 Score increased to about 0.358, suggesting that this model could explain around 35.8% of the variance in the transaction numbers. This improvement is a positive indicator, yet the R^2 Score still points to a significant portion of variance unexplained by the model.

Since a large proportion of contracts in the dataset have only one transactions, this data imbalance and the presence of other confounding factors may influence the performance of the models. To explore these possibilities further, we focused on the top 20 proxy contracts with the highest total transactions, as shown in Table 6. This approach aims to examine whether distinct patterns emerge among the most active contracts that could be better captured by our models.

In analyzing the top 20 proxy contracts, linear and Random Forest regression models were again employed, as shown in Figure 12. Figure 12 consists of two subfigures illustrating the performance of different models in predicting transaction numbers for the top 20 proxy contracts. Subfigure 12(a) displays the relationship between actual and predicted transaction numbers using a linear regression model, while Subfigure 12(b) shows the relationship for the random forest model. The x-axis represents the actual transaction numbers, while the y-axis denotes the predicted transaction numbers by the model.

Table 6 Top 20 proxies by total transactions

Proxy Address	Number of Versions	Total Transactions
0x1950bb2f0732a78b98adab118791b997a39bb29b	6	20,070
0x537907e5a708c6e2ac607df468ee49111f357596	19	10,016
0x1d3165b32897006935c482ffa0402ad68995ae4c	5	10,004
0x053d938a4d2a6df86d837d66a037444d7bacf3b9	2	10,001
0x2a5e45543ecf94de0cc1b574a41b73ff6d456e45	2	10,000
0x25f2f80d9a45b641bef25342a1b2a0ae48f78539	8	7,218
0x3cd5334eb64ebbd4003b72022cc25465f1bfcee6	2	2,597
0x71598610b7713d0321f70662a85a0f95df57db12	4	1,836
0xfc3bd18947b719e61b29d58d83bc55e99c7f5b31	2	1,612
0x59863022b862db70d3c52a1e6d6c0f778763a605	6	930
0x3570fed1bcfdda4e95cc7038d4d839c06da7e20d	2	918
0x4dff845c40d31b3a9b164ae1877901726fa06a2e	2	918
0x2a53c3fd708d671de8ac07a97456971f2244bd96	2	918
0x644b05a51630cd0152ce3bd3fe58bc7763756a2e	2	918
0x14f167abdba026c379142436a68d8979a342ecb5	3	765
0x50fda034c0ce7a8f7efdaebda7aa7ca21cc1267e	2	663
0x3b73c1b2ea59835cbfcadade5462b6ab630d9890	6	401
0x36e4ba1baec99e7f5950383beda589d3073c4d24	2	328
0x62faa8937f71b4896f9b250f675ff89a5f6875cc	50	280
0x5d30ad9c6374bf925d0a75454fa327aacf778492	16	238

The performance of these models was only moderately successful in explaining the variance in transaction numbers. The Random Forest model, in particular, showed a slight performance improvement, indicating some level of non-linear relationship but still pointing towards unaccounted factors influencing contract activity. However, the fact that a substantial portion of variance remains unexplained suggests that there are other factors influencing transaction numbers that are not included in the model. These could be factors intrinsic to the nature of smart contracts, user behaviors, market conditions, or other external variables not accounted for in the current model.

RQ4. How does smart contract upgrading impact the activity level of the contract?

Answer: The results suggests that the relationship between contract versions, their lifespans, and transaction numbers is complex and potentially influenced by factors beyond just the age and version number. These could include the contract's purpose, the nature of upgrades, user adoption rates, and external market conditions. The results also showed that the majority of the contracts are non active contract with only one transaction which is the creation transaction of the contract.

4.5 Case Study: Compound Comptroller (V2)

To strengthen our empirical findings, we turn to a detailed case study of Compound Comptroller (V2), one of the largest decentralized finance (DeFi) protocols, to showcase how upgradeable smart contracts are applied in practice.

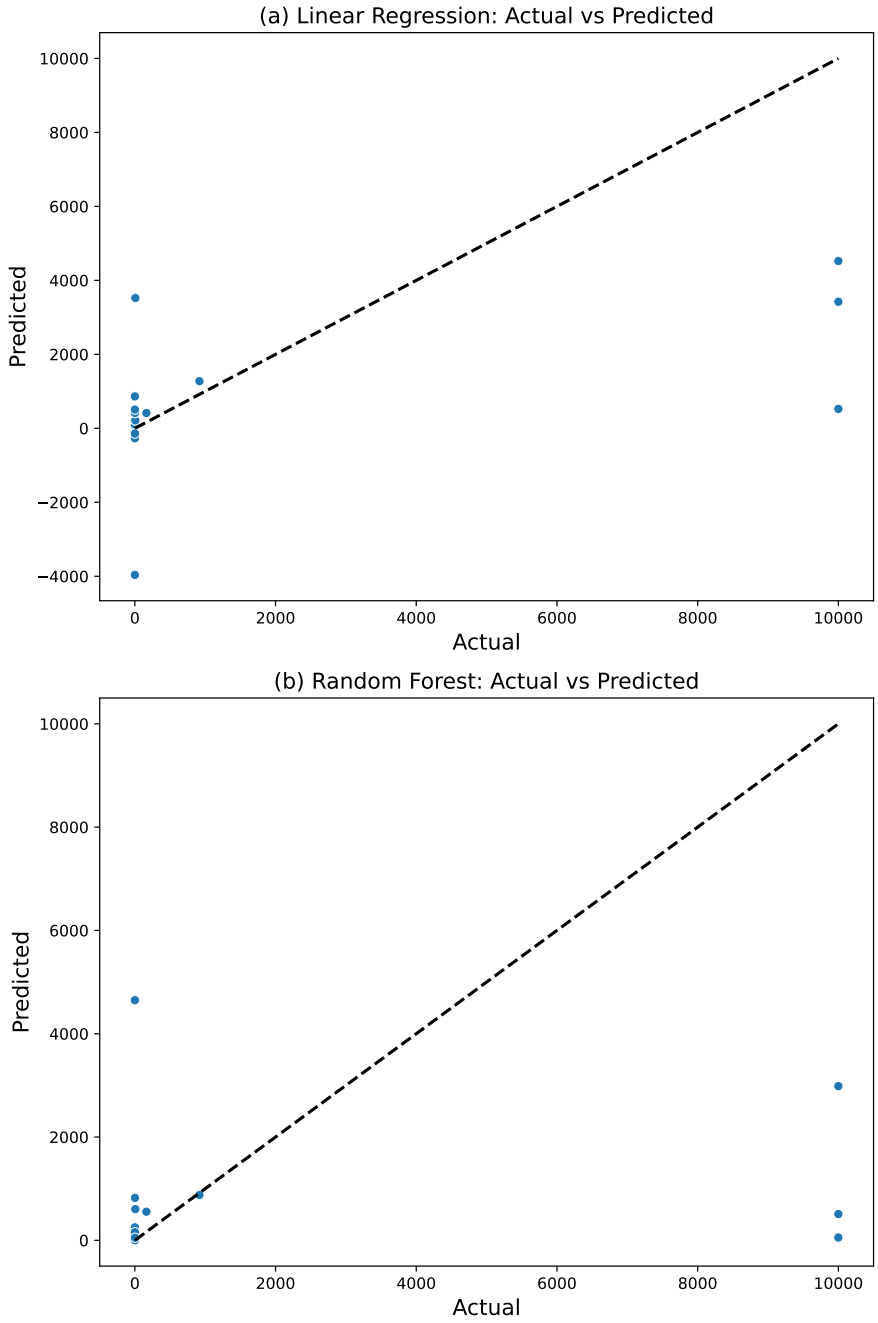

Fig. 12 Comparative analysis of regression models on smart contract data for top 20 proxy

Table 7 Compound V2 Historical Versions (RQ2) and Changes (RQ3)

Version	Comptroller Address	Post-upgrade Changes
V1	0x62f18c451af964197341d3c86d27e98c41bb8fcc	Initial Deployment
V2	0xf592ef673057a451c49c9433e278c5d59b56132c	New Feature, Bug Fix
V3	0x97BD4Cc841FC999194174cd1803C543247a014fe	Other
V4	0x9D0a0443FF4bB04391655b8cD205683d9fA75550	New Feature, Deleted Feature
V5	0xAf601CbFF871d0BE62D18F79C31e387c76fa0374	Other
V6	0x7d47d3f06A9C10576bc5DC87ceFbf3288F96Ea04	Gas Optimization, New Feature
V7	0xbe7616B06f71e363A310Aa8CE8aD99654401ead7	Gas Optimization, New Feature
V8	0x75442Ac771a7243433e033F3F8EaB2631e22938f	Gas Optimization, New Feature
V9	0x374ABb8cE19A73f2c4EFAd642bda76c797f19233	Other
V10	0x234B619B4f4E405665F7d94F2CE60c24256032B5	Gas Optimization
V11	0xbafe01ff935c7305907c33bf824352ee5979b526	Deleted Features, Bug Fix

Although we initially considered other high-profile protocols such as Aave[29] and Harvest Finance,[30] data limitations prevented us from fully analyzing their upgrade events. Aave does not emit upgrade events, making it challenging to track version history, and Harvest Finance lacks source code for the relevant versions. As a result, we focused on Compound,[31] where we had access to community insights and comprehensive version data, allowing us to explore the motivations, risks, and consequences of using upgradeable contracts within the DeFi ecosystem.

Compound is a major DeFi platform that enables users to lend and borrow cryptocurrencies, earning interest on their digital assets. The Comptroller contract, a core component of the protocol, manages risk, calculates interest rates, and distributes COMP, Compound's native governance token. At its peak in 2021, Compound held over eleven billion dollars in total value locked (TVL), underscoring its prominence in the DeFi landscape.[32]

The Comptroller is built on an upgradeable architecture using the EIP-1967 and Transparent Proxy pattern, allowing Compound to update the logic contract without affecting user balances or requiring new contract addresses. The Transparent Proxy separates administrative functions from user interactions, with administrative tasks (such as upgrades) managed by a designated ProxyAdmin. The Comptroller contract, identified by proxy address 0x3d9819210a31b4961b30ef54be2aed79b9c9cd3b, has undergone multiple upgrades to introduce new features, fix bugs, and optimize gas costs. In our research, we tracked these upgrades through the `NewImplementation` event emitted during each upgrade.

The version history of Compound's Comptroller contract is summarized in Table 7, highlighting post-upgrade changes (RQ3).

[29] `https://aave.com`

[30] `https://harvest.finance`

[31] `https://compound.finance/`

[32] `https://defillama.com/protocol/compound-finance#information`

4.5.1 Reasons for Upgrades

We examined Compound's GitHub repository[33] and official governance proposals[34] to understand the reasons behind these upgrades. By mapping commits to contract versions (using addresses from commit details), we classified upgrades and compared them to the categories in Table 7.

Regarding feature modification, six versions introduced changes to the Comptroller's functionality. For instance, Version two introduced the *Timelock Administrator* to decentralize control, ensuring that protocol changes were delayed, giving the community time to review and audit governance proposals. Version two also introduced the *Pause Guardian* feature, which allows pausing critical operations like borrowing or liquidation during emergencies to protect user funds.

Version five introduced the *COMP Distribution System* (Flywheel), incentivizing user participation by distributing COMP tokens based on lending and borrowing activity. This upgrade significantly boosted Compound's TVL by encouraging more engagement. However, Version nine introduced a bug while refining the COMP distribution system, which led to the over-distribution of approximately one hundred million dollars in COMP tokens[35].

The governance structure delayed addressing the security issue introduced in Version nine due to a seven-day *Timelock*. Version ten responded by restricting the withdrawal of over-accrued COMP to only those users who had not interacted with the affected markets[36]. Version eleven fixed the issue entirely, restoring the COMP rewards system[37].

In addition to security fixes, Compound's upgrades focused on optimizing gas costs. Versions six and seven, for instance, introduced significant gas savings for actions like claiming COMP rewards and improving the protocol's overall efficiency.

Upon comparing the versions from the GitHub commits, we identified some variations in our initial classification. For example, the changes in Version three involved an update to the interest rate model, which affected minor code components. Our original analysis classified this version as "Other," due to the limited scope of these changes. The update primarily addressed interest rate behavior and verification tests, which did not modify entire functions or introduce substantial new features. Similarly, Version five involved minor fixes, including minor code quality improvements. The upgrade was intended to address second-order issues related to COMP distribution, but the actual code changes were minimal, and no substantial functionality was added or removed.

[33] `https://github.com/compound-finance/compound-protocol`

[34] `https://compound.finance/governance`

[35] `https://sharkteam.org/report/analysis/20211012001A_en.pdf`

[36] `https://compound.finance/governance/proposals/64`

[37] `https://compound.finance/governance/proposals/65`

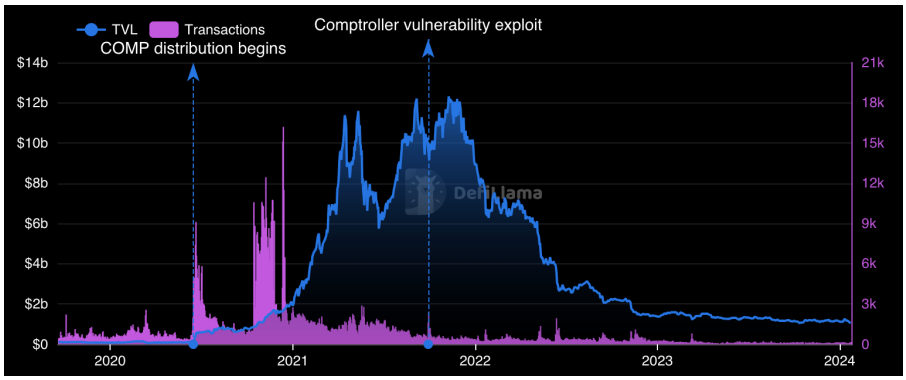

Fig. 13 Total Value Locked (TVL) and transaction volume for Compound V2

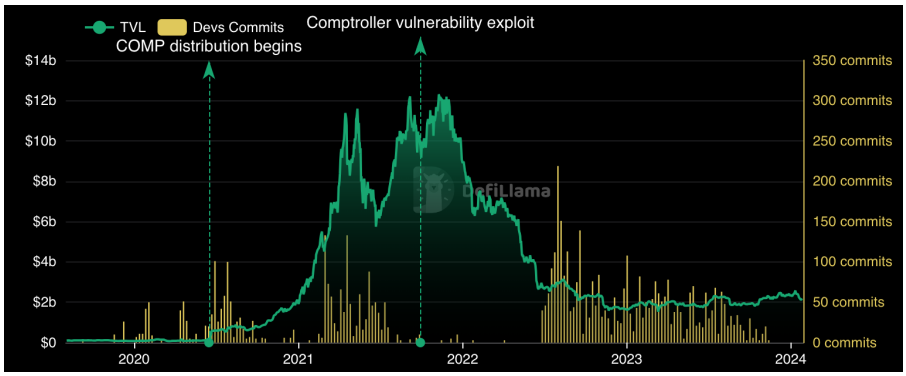

Fig. 14 Total Value Locked (TVL) and developer activity (measured by commits) for Compound Finance

4.5.2 Impact on Stakeholders

The security issue had a significant impact on user engagement, developer activity, and overall protocol health. As shown in Figure 13 (sourced from DeFiLlama[38]), there was a sharp drop in transaction volume and TVL following the exploit in 2021. Even after the fix in version eleven, transaction volumes did not return to pre-incident levels, indicating a prolonged loss of user trust.

Figure 14 (sourced from DeFiLlama[39]), highlights the correlation between developer activity (commits) and protocol engagement. Developer engagement peaked in 2020 and early 2021, coinciding with Compound's growth in transaction volume and TVL. After the Comptroller exploit, the commits to Compound declined, likely due to the development focus shifting to Compound V3, which required substantial architectural changes.

[38] `https://defillama.com/protocol/compound-v2?transactions=true`

[39] `https://defillama.com/protocol/compound-finance?devCommits=true`

[39]

Current Status According to Compound's founder, Robert Leshner, the V2 architecture posed significant risks, leading to the introduction of Compound V3. In V2, if the value of a single asset dropped to zero, it could drain the entire protocol. V3 mitigates this risk with a new collateral structure designed to protect the protocol from individual asset failures.

The proxy address for Compound V3 is *0x1EC63B5883C3481134FD50D5DAebc83Ecd2E8779*. However, unlike V2, V3 does not emit upgrade events, making it difficult to trace when upgrades occur. Based on our classification in RQ1, we categorize Compound V3's proxy as an *Other Upgradeable Proxy Contract*, as it follows the Transparent Proxy pattern but deviates from EIP-1967 by not using the standard storage slot and not emitting upgrade events.

4.5.3 Broader Implications

One critical aspect influencing Compound's usability, especially following the security issue, is its governance structure. Compound employs a decentralized governance model where token holders (COMP) can propose, debate, and vote on protocol changes. This system promotes transparency and community involvement, allowing users to have a say in the protocol's direction, which is crucial in the DeFi space. The governance system ensured that Compound's users felt a degree of control over the protocol's upgrades and modifications, contributing to a sense of ownership and responsibility for the system's health. This was also enhanced by the platform's historical commitment to security audits, which further reinforced user trust. Even though there was no specific audit for the version that introduced the security issue, Compound had undergone nine independent audits for previous versions.[40] These factors likely played a role in encouraging more users to participate in governance decisions and kept the user base engaged with the platform (as shown in Figure 13, where the user transaction increased after introducing the governance mechanism).

However, governance also introduced challenges, as evidenced by the Version nine bug that over-distributed $100 million in COMP tokens. In that instance, the Timelock delay hindered a rapid response to the bug, demonstrating a potential trade-off between decentralization and operational efficiency.

Key takeaways from the case study

A key takeaway from this case study is that governance mechanisms, security audits, and transparency significantly influence user trust and engagement. While upgrades can improve protocol functionality and security, they can also disrupt activity levels if not carefully managed. In Compound's case, the delayed response due to decentralized governance, combined with security vulnerabilities, led to a sharp and lasting decline in transaction activity. This complements our findings in RQ4, where we

[40] `https://docs.compound.finance/v2/security/`

concluded that factors beyond contract age or version, though not explicitly identified, play a critical role in the activity levels of a contract.

5 Discussion

This section further discusses the RQs results and gives insights into smart contract upgradeability.

5.1 RQ1: Prevalence of Upgradeable Smart Contracts

Based on the analysis of the prevalence of upgradeable smart contracts in RQ1, we analyzed the community discussions about these standards to reason about the adoption of one standard compared to others. We observed that only three standards, Diamond (EIP-2535), Proxy Storage Slots (EIP-1967), and OpenZeppelin, remain actively used. The transparent standard (EIP-1538) was withdrawn with subsequent modification, and the Diamond standard was introduced, indicating a refinement in upgradeable contract approaches. Meanwhile, UUPS (EIP-1822) and EIP-897 standards are stagnant, having not seen significant activity in the past six months. From the official discussion of the UUPS (EIP-1822) standard,[41] we also identified a security concern that revolves around the proxiableUUID mechanism. This mechanism is designed to ensure compatibility between a proxy and its implementation contracts by utilizing a unique identifier. The core of the security issue lies in the potential for proxies to mimic the logic intended for implementation contracts, thereby creating a vulnerability during the upgrade process. This vulnerability poses significant risks, such as creating loops of recursive delegation that never reach actual implementation logic, leading to gas depletion and failed transactions. Moreover, it might result in the execution of unintended or malicious logic, compromising contract integrity and potentially leading to the irreversible loss of contract functionality and assets. This security concern is likely a contributing factor to the observed decline in UUPS (EIP-1822) usage shown in Figure 6. The discussion of this issue was in March 2021, and its alignment with the pattern's decline in the figure strongly suggests a correlation between the security concerns and the reduced adoption of this standard in mid-2021.

While analyzing the popular EIP-1967 standard, we observed that most EIP-1967 proxies are Transparent Proxies, though other proxy patterns also use EIP-1967. The Transparent Proxy pattern is widely adopted due to its clear separation of admin and user roles: the admin manages upgrades, while users interact with the contract's business logic. This separation minimizes the risk of users accidentally triggering admin functions, ensuring smoother contract operation. Only the admin can invoke upgrade functions, preventing unintended interactions with the admin logic. It is important to note that

[41] https://ethereum-magicians.org/t/eip-1822-universal-upgradeable-proxy-standard-uups/2842/30

Transparent Proxy, introduced by OpenZeppelin, follows the EIP-1967 standard and differs from EIP-1538, which was withdrawn.

In contrast, UUPS (EIP-1822) handles upgrades within the implementation contract itself, reducing the risk of function clashes but introducing the possibility of accidentally removing upgradeability. While this feature can be useful for developers aiming for contract immutability, it also presents risks if not managed carefully. Hence, EIP-1967 with Transparent Proxy logic is likely more popular due to its simplicity and clear separation of roles. Separating upgrade functions from user operations provides a straightforward and secure method for managing upgrades, making it an appealing choice for developers who require flexibility without the risk of accidentally disabling upgrades.

From the results of RQ1 in Figure 5, we focused on a subset of smart contracts whose proxy types are *Other upgradeable proxy contract*, aiming to classify and understand their underlying structures. We identified three uniform patterns among a subset of these contracts through a detailed analysis:

- CErc20Delegator: Identified in 571 contracts, this pattern is part of the Compound protocol and is unique for its detailed constructor, which initializes DeFi-specific parameters (such as the underlying asset, interest rate models, and initial exchange rates) and sets the implementation contract. The contract emits a *NewImplementation* event when a contract is upgraded, including the old and new addresses of the contract.
- Proxyable/ProxyERC20: Designed for ERC20 token upgradeability, this pattern extends the *Owned* contract (for ownership and access control management) and is marked as 'payable.' The constructor of this contract type ensures that the contract cannot be instantiated directly, enhancing its security. The *TargetUpdated* event is used to signal updates to the proxy's target, providing transparency for contract upgrades. This pattern was found in 19 contracts from the *Other upgradeable proxy contract* category.
- Custom Proxy Contracts on Polygon: This proxy, appearing in 31 contracts, features a *ProxyUpdated* event and suggests the deployment of custom proxy contracts on the Polygon network. Although they align with ERC-897's DelegateProxy pattern, they are recognized under a distinct interface, *IERCproxy*.

While we were able to identify three uniform patterns among these contracts through a detailed analysis, the reasons behind developers' preference for using less standardized or custom hybrid proxies remain unclear. It is possible that these choices are driven by specific project needs or ecosystem preferences, but our current data does not provide direct evidence for these motivations. This highlights an area for future investigation to better understand the decision-making process behind adopting custom or less standardized proxy patterns.

The significance of these findings lies in the implications for different stakeholders within the smart contract ecosystem. Developers can use this information to choose upgrade standards that are not only popular but also secure, such as the well-adopted EIP-1967, which addresses function clashes through Transparent Proxy's clear separation of responsibilities. This knowledge helps

them avoid security vulnerabilities, such as those identified in UUPS (EIP-1822). Platform providers can focus on supporting these common standards, ensuring their platforms can accommodate the most widely used upgrade mechanisms. Meanwhile, security auditors can prioritize auditing standards like EIP-1967 and Transparent proxy, where vulnerabilities are less likely but security remains crucial due to their prevalence.

5.2 RQ2: Historical Versions of Smart Contracts

A limitation of our approach is its reliance on event-based detection to track smart contract upgrades. Our analysis observed that 12.57% of upgradeable proxies did not emit upgrade-related events. To understand this more deeply, we conducted further analysis of the types of upgradeable proxies that do not emit events. Our findings revealed that 99.98% of proxies based on EIP-1822 (UUPS) and 99.41% of proxies based on EIP-897 (DelegateProxy) did not emit any upgrade events. Similarly, 71.8% of OpenZeppelin-based contracts also did not emit upgrade-related events. Even for EIP-1967 (Proxy Storage Slots), which explicitly recommends emitting events during upgrades, 9.77% of the contracts did not emit such events. These three approaches, EIP-1822, EIP-897, and OpenZeppelin, do not explicitly require upgrade events in their standards, contributing to the lack of emitted events. This suggests that event-based detection, while useful, may not capture all upgrades, particularly in cases where the standards do not mandate event emissions. The absence of upgrade events in proxies adhering to these standards limits the scope of event-based upgrade detection, potentially leading to underestimating the actual upgrade frequency, which, in our findings, is 0.34%. These undetected upgrades from the 12.57% of contracts may result in a higher true upgrade rate. However, this limitation is specified to this subset of contracts, as 87.43% of upgradeable proxies did emit events, meaning the observed 0.34% upgrade rate is still notably low.

This observation challenges the presumed advantage of upgradeability, highlighting a disparity between its theoretical capabilities and actual application in practice. Two primary factors could explain this phenomenon: the contracts might be of high initial quality due to thorough auditing and testing before deployment, or the complexity and risks associated with upgrading might deter their application.

Firstly, the low upgrade rate may be linked to the high initial quality of smart contracts, which are carefully audited and tested before they are deployed. Auditing typically involves reviewing the contract's code to identify vulnerabilities and ensure it functions securely. This process includes both automated tests and manual inspections by security experts. Developers and project managers can rely on this rigorous preparation to minimize the need for changes after deployment. For instance, as shown in our Compound V2 case study, the Compound protocol underwent nine security audits for different versions of the Comptroller contract before upgrades were deployed. However,

even large projects like Compound and Aave, which go through thorough security audits, still apply upgrades to fix issues or add new features. This shows that while auditing can reduce the pressure to upgrade frequently, some contracts, especially those with more complex needs, still require updates to stay secure and efficient.

Secondly, the complexity and risks of the upgrade process itself can discourage frequent upgrades. Upgrading a contract requires careful planning to ensure compatibility with existing functions and to avoid new vulnerabilities. Each upgrade introduces the risk of unforeseen issues, making the process similar to the caution and thoroughness needed during the original deployment. For example, in the Compound V2 case study, the upgrade to V9 introduced a bug in the COMP distribution logic that caused a significant financial loss. Despite thorough audits, the governance model's seven-day Timelock prevented an immediate fix, illustrating how the risks involved in upgrades can sometimes outweigh the benefits, further discouraging frequent updates. Security auditors play a key role here, as they need to verify that any upgrades maintain the contract's integrity without introducing new risks. Investors may see a low upgrade rate as a sign of stability, but they should also be aware that it could signal a reluctance to address new technological developments or emerging security concerns. Regulators might view low upgrade activity as a risk if contracts are not updated to meet changing legal requirements or compliance standards.

Another factor influencing the low upgrade frequency is the relatively short period since many upgradeable proxies were deployed, particularly after 2022. The data in our study was collected up to July 2023, meaning that several of these proxies have only been operational for a short time. This limited timeframe may not have provided sufficient opportunity for these contracts to require upgrades, especially if they were initially deployed with high-quality code and had undergone rigorous audits before launch. Additionally, many developers may still be in a monitoring phase, observing the performance of these contracts under real-world conditions before initiating any upgrades.

The impact of upgrades on contract quality is complex. On one hand, upgrades allow developers to fix problems and improve functionality. On the other hand, they also carry risks that can compromise the contract if not done carefully. For developers and project managers, deciding whether to upgrade requires weighing the benefits of improved security and functionality against the potential risks and challenges involved in the process.

5.3 RQ3: Analyzing Post-Upgrade Changes in Smart Contracts

Based on the results from RQ3, we investigated the relationship between the availability of source code and the upgrade frequency of smart contracts. Our analysis divides smart contracts into two groups based on source code availability, leading to a comparative study of how this factor influences the evolutionary trajectory of these contracts.

Contracts with available source code exhibit a higher mean number of versions than those without. Specifically, the mean version count for contracts with accessible source code is approximately 2.71, compared to 2.00 for contracts without source code. This statistically significant disparity suggests that open-source contracts undergo more frequent updates. This trend aligns with established open-source practices in traditional software development, where transparency, community involvement, and iterative development are key to rapid issue resolution and continuous improvement.

While these principles are generally applicable to all open-source software, our findings show that they manifest similarly in the context of smart contracts. For instance, in our Compound V2 case study, the open-source nature of the contract allowed the community to propose and validate changes, helping to identify potential vulnerabilities, such as the bug found in the COMP distribution logic. This issue was openly discussed within the community, and a subsequent upgrade addressed the problem, illustrating how smart contracts benefit from community scrutiny and more dynamic development cycles, just like other open-source projects.

5.4 RQ4: Upgradeability Impact on Contract's Activity level

Our study aimed to investigate the relationship between smart contracts version activity level, measured by total transactions, and their upgrade frequency. Since there is no linear relationship between the number of transactions and the age of the contract, we focused on comparing the upgrade intervals of the most actively used contracts with those less active. This approach was intended to identify if activity level influences upgrade practices, as we have noticed many smart contract updates within minutes. We divided the proxy contracts into two groups for analysis: the top 20 proxies by transaction volume and the remaining proxies, excluding the latest version of each, to consider only complete upgrade cycles. The top 20 proxies demonstrated an average upgrade interval of approximately 609 days (20.3 months), while the remaining proxies had a shorter interval of about 521 days (17.4 months).

This differential suggests that more actively used contracts are upgraded less frequently. For developers and project managers, this insight could inform strategies for balancing system stability with the need for upgrades in high-transaction environments. Fewer upgrades might reduce the risk of disrupting users in heavily utilized contracts. At the same time, this might reflect a strategic decision to avoid unnecessary changes in critical systems. End users benefit from fewer disruptions but may expect transparency about when and why upgrades are made, especially in high-stakes contexts where contract reliability is crucial.

5.5 Smart Contract Upgradeability

The empirical findings highlight several key observations about the prevalence and usage of upgradeable smart contracts, offering important insights into the "to upgrade or not to upgrade" dilemma faced by developers in the blockchain ecosystem. While only 3% of all smart contracts in our dataset are upgradeable proxies (as identified in RQ1), their significance becomes apparent when considering other findings. For instance, only 0.34% of upgradeable proxies have undergone upgrades post-deployment (RQ2), and 90.16% of smart contract versions have processed a single transaction. This highlights a cautious approach in the community, where developers may prioritize immutability and stability over the complexities of performing upgrades. The decision-making process around smart contract upgradeability reveals a complex landscape in blockchain development. Developers and stakeholders must weigh several factors: addressing critical vulnerabilities, adapting to new requirements, and maintaining the core principle of immutability. This balance between innovation and stability is essential to understanding the upgrade dilemma, reflecting broader challenges in the evolution of blockchain technologies. As our study shows, this dilemma is particularly prominent in projects where upgrades are technically possible but rarely applied in practice.

However, despite the low percentage of upgradeable proxies and the infrequency of upgrades, the importance of understanding smart contract upgradeability cannot be understated. As highlighted in RQ1, these upgradeable proxies, though representing only 3% of all smart contracts, collectively hold over $3.3 billion in Total Value Locked (TVL). Our Compound V2 case study further emphasizes the critical role of upgradeability in high-value projects. The project went through multiple upgrades to enhance security and functionality. These upgrades were essential to its stability and growth, showcasing how upgradeability is crucial when large amounts of value are at stake.

Given the high stakes involved and the significant economic value held by upgradeable proxies, it is essential to study this topic in more detail. The fact that upgradeable proxies are being used in high-impact projects demonstrates their strategic importance, even if upgrades are not frequent. Moreover, since smart contract upgradeability is a relatively new area of study, these findings lay a foundation for further research, helping developers and researchers better understand upgrade practices and the risks and benefits associated with them.

6 Related Work

There are only a handful of studies focused on the upgradeability of smart contracts. This section presents and discusses these existing studies for smart contract upgradeability.

Salehi et al. (Salehi et al., 2022) analyzed and evaluated smart contract upgradability patterns. The authors presented a framework for measuring the number of upgradeable Ethereum contracts which utilize certain well-known

upgradeable proxy patterns. Furthermore, they have analyzed how access control is implemented over smart contract upgradeability.

Bui et al. (Bui et al., 2021) proposed a Comprehensive-Data-Proxy pattern to upgrade smart contracts while enhancing security resilience and scalability. The authors investigated three popular Ethereum attacks that affect smart contract upgradeability: cross-function Reentrancy, typical Reentrancy, and DAO attacks. The presented pattern improves resilience against such attacks compared to the previous upgrading approaches.

Chen et al. (J. Chen, 2020) introduced a deep learning-based method to detect security issues in the updated version of a destructed smart contract. A contract can be destroyed on Ethereum only by using the Selfdestruct function, which transfers all the Ethers on the contract balance to the contract owner. The authors compare the historical version of destructed contracts and investigated whether security issues were detected in the destructed contracts.

Fröwis et al. (Frowis & Bohme, 2022) investigated the impact and evaluated the CREATE2 instruction adoption in Ethereum smart contracts. The CREATE2 instruction allows the contract to be modified after deployment on a given address. Furthermore, the authors identified several use cases and attack vectors for the CREATE2 instruction.

Bodell III et al. (Bodell III et al., 2023) conducted research on upgradeable smart contracts using proxies to investigate the current situation and identify security issues. The authors created a comprehensive classification of proxy-based USCs that can distinguish their behaviors based on both structural and semantic characteristics. They also developed USCHUNT, a static analysis framework for detecting and examining USCs, and used it to analyze over 800K smart contracts across eight popular blockchains. Their findings include 11 distinct USC design patterns and six types of security and safety concerns.

Ebrahimi et al. (Ebrahimi, Adams, Oliva, & Hassan, n.d.-a) investigated the proxy pattern (whether upgradeable or non-upgradeable) using a dataset containing 50 million smart contracts. Using a behavioral detection technique, they discovered that over 14% of the contracts are active proxies. The study also revealed an increasing trend in using proxy contracts throughout Ethereum's history, highlighting their significant role in enhancing modularity. Moreover, the authors identified 12 distinct creational patterns for deploying proxies (upgradeable and non-upgradeable), which we categorized as off-chain and on-chain styles based on where the deployment script operates. However, this classification does not specifically focus on upgradeable proxy contracts and deviates from established standard classifications. Moreover, Ebrahimi et al. (Ebrahimi, Adams, Oliva, & Hassan, n.d.-b) introduced UPC Sentinel, a two-layer algorithm that combines static and dynamic bytecode analysis to accurately detect Upgradeability Proxy Contracts (UPCs) on Ethereum. Their tool achieves an impressive recall of up to 98.4% and perfect precision in empirical evaluations. Nevertheless, a shortcoming of UPC Sentinel is its inability to detect inactive proxies, those that have not delegated any calls since deployment. This limitation can obscure upgradeable contracts that are dormant or have yet to be fully utilized in their proxy roles.

Huang et al. (Huang et al., 2024) examine upgradeable smart contracts by focusing on proxy patterns and metamorphic contracts. They develop a bytecode-based classification model that analyzes opcode frequencies, achieving a high F1-score of 99.37% for detecting proxy-based upgradeable contracts. However, their model relies heavily on the assumption that opcode frequency is a reliable indicator of upgradeable contracts, which may not hold true in all cases. For example, the location of specific opcodes, rather than their frequency, can be critical for accurate detection.

Li et al. (Li, Yang, Chen, Tang, & Gao, 2024) propose USCDetector, a tool designed to identify upgradeable smart contracts on Ethereum without relying on source code. By analyzing bytecode in combination with known function selectors such as `upgradeTo` and `setImplementation`, they detect common proxy patterns with a precision of 96.26%. Furthermore, their tool extends its detection to non-proxy upgrade patterns, including data separation and strategy patterns, which split logic and data across different contracts. A notable limitation of USCDetector is its reliance on predefined function selectors, which may limit its ability to detect upgrade logic implemented using unconventional function names, thus potentially reducing its overall detection coverage.

Liu et al. (Liu et al., 2024) present UpCon, an innovative framework that detects and characterizes proxy-based upgradeable smart contracts on Ethereum. UpCon operates at the source code level and leverages verified contracts from platforms such as Etherscan for accurate detection. Their binary search algorithm optimizes the detection of historical implementation addresses, reducing the complexity from linear to logarithmic time. However, UpCon's reliance on source code limits its scope, as it cannot analyze unverified contracts that do not provide access to source code.

Table 8 illustrates the comparison between previous studies and our empirical study regarding smart contract upgradeability. The focus of existing studies on smart contract upgradeability has primarily been detecting upgrading approaches, particularly proxy-based methods. However, only two studies (Salehi et al., 2022; Bodell III et al., 2023) provided insights on the prevalence of upgrading proxy patterns. In this study, we focused beyond the prevalence of upgrading contracts (RQ1), where we analyzed smart contract versions (RQ2), post-upgrade modifications (RQ3), and the impact of upgrading on the adoption of smart contracts (RQ4). Nevertheless, there are aspects that this study did not cover which were discussed in previous studies, such as the security issues related to the existing proxy approaches (Bodell III et al., 2023) as well as the access control analysis of these contracts (Salehi et al., 2022; Ebrahimi et al., n.d.-a). While these are interesting topics, they were not within the scope of our empirical study on smart contract upgradeability. Table 9 provides an overview of studies focused on detecting upgradeable proxy contracts. It highlights key aspects such as the analysis level (whether bytecode, or source code were analyzed), the number of upgradeable proxies detected, whether non-active proxies were identified, and the types of proxies detected. Additionally,

Table 8 Comparison of studies on smart contract upgradeability

Study	Smart Contract Upgrading Approach	Smart Contract Versions	Post-Upgrade Analysis	Security Issues	Access Control Analysis	Activity Level Analysis
(Salehi et al., 2022)	Proxy				✓	
(Bui et al., 2021)	Proxy					
(J. Chen, 2020)	Self-destruct function					
(Frowis & Bohme, 2022)	CREATE2 instruction					
(Bodell III et al., 2023)	Proxy			✓		
(Ebrahimi et al., n.d.-a)	Proxy				✓	
(Huang et al., 2024)	Proxy		✓		✓	
(Liu et al., 2024)	Proxy	✓	✓	✓		
(Li et al., 2024)	Proxy Data seperation Strategy Pattern Metamorphic Contract	✓		✓		
Our Study	Proxy	✓	✓	✓		✓

Table 9 Overview of Studies Focused on Detecting Upgradeable Proxy Contracts

Study	**Analysis Level**	**# Upgradeable Proxies Detected**	**Non-Active Proxies Detected?**	**Types of Proxies Detected**	**Historical versions**	**Detection methods for versions**
(Salehi et al., 2022)	Bytecode and Transactions	8225	No	3		
(Bodell III et al., 2023)	Source Code	8815	Yes	11		
(Ebrahimi et al., n.d.-b)	Bytecode and Transactions	∼3000	No	9		
(Liu et al., 2024)	Source Code	44,282 (upgradable and non upgradeable)	Yes	-	973	Contracts with implementation function
(Li et al., 2024)	Bytecode and Transactions	43,650	Yes	-	4,692	Functions
Our Study	Bytecode	∼1300000	Yes	7	14,990	Events

the table includes information on whether historical versions of the contracts were tracked and the methods used to detect these versions.

6.1 Evaluation of the Detection Approach for Upgradeable Proxies

We focused our comparison on USC HUNT, as both the data and the tool are publicly available, unlike other approaches discussed in Table 8, which lack accessible datasets or tools for direct evaluation. However, comparing USC HUNT with our approach presents challenges due to differences in methodology: USC HUNT relies on source code analysis, while our tool focuses on bytecode. Many contracts in our dataset are unverified, which prevents us from using the USC HUNT tool to evaluate our classifications directly. Consequently, we based our evaluation on the subset of contracts classified by USC HUNT and assessed their bytecode to determine discrepancies between the two approaches.

We compared our proxy detection approach with the USCHunt tool using two datasets in the USCHunt GitHub repository. The first dataset[42] contains

[42] `https://github.com/USCHunt-Anon/USCHunt/blob/master/study/artifacts/upgrade_counts.json`

975 contracts flagged as upgradeable proxies, identified based on common upgradeability patterns, such as the presence of `delegatecall` and setter functions for modifying the implementation contract's address. The second dataset[43] is a ground truth dataset used to evaluate the USCHunt tool, which includes both forward and upgradeable proxies, totaling 958 contracts. After combining both datasets and removing duplicates, we had 1,166 unique contracts for our evaluation.

Upon comparison, we identified several misclassifications. To understand these discrepancies, we manually analyzed the contracts using their source code, decompiled bytecode, and the bytecode itself to pinpoint the root causes. Out of the 1,166 contracts, we found differences in 37.9% (442 contracts), where USCHunt classified them differently from our approach. Specifically, some of these contracts were classified as upgradeable by USCHunt, but our method identified them as either non-proxies or forward proxies. This subset includes 219 contracts classified as non-proxies and 202 contracts classified as forward proxies by our method. Additionally, 8 cases were classified as upgradeable by our approach, but USCHunt identified them as non-upgradeable proxies. Lastly, 12 contracts could not be definitively classified by our approach due to failed decompilation caused by automation issues.

Misclassification Analysis For the 219 contracts classified as non-proxies (18.78%), further analysis revealed that these contracts contained `delegatecall` in the source code provided on Etherscan, which likely led USCHunt to flag them as upgradeable proxies. However, the `delegatecall` was absent in the bytecode of the deployed contract at the analyzed address. This discrepancy arises because developers included additional contracts in the verified source code on Etherscan that were implementation contracts without any upgradeability functionality themselves. The actual deployed contract, which serves as the logic contract, did not contain `delegatecall`, and a separate contract managed the proxy functionality. Our bytecode-based approach accurately classified these contracts as non-proxies, as the deployed bytecode lacked evidence of delegation.

Regarding the 202 contracts classified as forward proxies (17.32%), USCHunt flagged these as upgradeable proxies, but our analysis identified them as forward proxies. The key distinction was that the implementation address in these contracts was immutable post-deployment, with no setter functions or mechanisms to alter the implementation address. As a result, while these contracts employed the `delegatecall` function, they did not support upgrading the implementation address, functioning instead as forward proxies with fixed implementations.

These patterns align with observations made in other studies (Huang et al., 2024; Ebrahimi et al., n.d.-b), which reported similar findings where source code-based analysis led to false positives due to the inclusion of additional

[43] `https://github.com/USCHunt-Anon/USCHunt/blob/master/study/data/ground_truth/Upgradeability_Ground_Truth.xlsx`

Table 10 Evaluation Metrics for USCHunt and Our Approach

Metric	USCHunt	Our Approach
Precision	58.97%	100%
Recall	98.69%	97.90%
F1 Score	73.83%	98.94%

or unrelated contracts. However, the analysis data from these studies are not publicly available, limiting direct comparisons. In 8 cases (0.7%), our approach classified the contracts as upgradeable, while USCHunt identified them as non-upgradeable proxies. Upon analyzing both the decompiled bytecode and the available source code, we found that these contracts were indeed upgradeable, as the implementation address could be modified in a separate function. USCHunt acknowledges this low rate of false negatives, and these cases represent the mismatches we identified.

Our approach encountered limitations for the 12 contracts (1.0%), and we were unable to classify them definitively due to failed decompilation caused by automation issues. These failures occurred when automating the feeding of multiple bytecodes to evaluate the approach. However, when tested individually, no failures were encountered, indicating that the issue was related to automation rather than our detection approach or the decompiler. Nevertheless, we recorded these cases as false negatives since the automated classification was incorrect (or unknown).

The detailed classification results for USCHunt and our approach, including which contracts were flagged as upgradeable, non-proxies, or forward proxies, are available in a CSV file in the project repository.[44]

Evaluation Results The evaluation of our method versus USCHunt is summarized in terms of precision, recall, and F1 score, as shown in Table 10. USCHunt is more susceptible to false positives, primarily due to its reliance on source code that may include contracts not directly relevant to the deployed bytecode. In contrast, our bytecode-based approach provides higher precision by filtering out non-upgradeable implementation contracts and forward proxies, as reflected in Table 10. However, our method also has limitations, particularly when dealing with large-scale automation.

7 Threats to Validity

This section outlines potential threats to the validity of our study, classified into four groups: internal validity, external validity, construct validity, and conclusion validity.

[44] https://drive.google.com/drive/folders/1xlY7suki5TqiOF7mipKspfBbTb888-iz?usp=sharing

7.1 Internal Validity

A significant concern in our study is the limitations associated with using automated tools for detecting vulnerabilities. These tools could potentially produce false positives or miss certain vulnerabilities, impacting the accuracy of our findings. To mitigate this, we applied a majority rule approach, where a vulnerability is considered valid only if detected by multiple tools within the SmartBugs suite (Soud, Qasse, Liebel, & Hamdaqa, 2023; Yashavant, Kumar, & Karkare, 2022). While this strategy helps reduce the occurrence of false positives, it does not fully eliminate them. We acknowledge that the vulnerabilities detected by SmartBugs may not encompass all real-world scenarios, particularly those related to newer or less common contract behaviors (Zhang, Zhang, Xu, & Lin, 2023), which remains a limitation of the current automated analysis tools.

Furthermore, SmartBugs may not specifically target vulnerabilities related to upgradeable proxy contracts (such as storage collisions). While some tools, like ConFuzz, can detect issues such as unsafe delegate calls, this is beyond the scope of our study, which focuses on analyzing different versions of smart contracts rather than specific vulnerabilities associated with upgradeable proxies.

Another threat to internal validity stems from identifying and categorizing smart contracts and their versions. While our approach does not involve manual labeling, which could introduce subjective biases, there remains a possibility of inaccuracies in the automated identification and classification process. We have extensively tested and validated each step of our methodology to counter this risk. This testing serves not only as a check on the accuracy of our processes but also as a validation of the methods themselves. It provides us with confidence in the applicability of our findings, ensuring that our conclusions are based on reliable and correctly categorized data.

7.2 External validity

Our study focuses on Ethereum smart contracts, which may limit our findings' generalizability to other types of smart contracts or blockchain platforms. While the study aimed not to generalize all smart contracts but to provide insights into the impact of upgrading on contracts' security, we acknowledge this limitation. The thorough testing of sample data ensures a solid foundation for our conclusions within the scope of Ethereum-based contracts.

Additionally, our study specifically focuses on the proxy pattern among upgradeable smart contracts, acknowledging that other patterns exist but are not the focus of our research. We have excluded other types as they do not exhibit unique characterization to distinguish them. Moreover, some types have minimal prevalence, such as data separation, which represents only 0.0007% of the dataset. Threats to validity may arise from the limited scope of our study, as we only consider the proxy pattern among upgradeable smart contracts due to its significant role and the concentrated attention it receives within the

smart contract development community. Moreover, focusing on proxy patterns allows for a more detailed and impactful analysis of the most commonly employed upgrade techniques in the Ethereum platform. However, this narrow focus may limit the generalizability and applicability of our findings to other patterns or platforms.

A potential threat to external validity is the representativeness of our sample of 5,000 upgradeable proxy smart contracts in capturing the characteristics of the entire population of 1.3 million proxy contracts. To mitigate this, we used a stratified random sampling approach based on the deployment year and proxy type, ensuring a proportional representation of each stratum. We also conducted a Chi-square test for goodness of fit, confirming that the sample's distribution is statistically indistinguishable from the full dataset (Chi-square statistic = 1.912, p-value = 1.0). This sampling approach minimizes bias, thereby supporting the reliability of subsequent analyses based on the sample data.

7.3 Construct Validity

Our approach relies on detecting smart contract upgrades through emitted events, which may not cover all upgrade scenarios. As our analysis showed, approximately 12.57% of proxy contracts did not emit any relevant events, leading to potential gaps in our detection. While we considered other methods, such as tracing function calls, these methods were found to be impractical due to issues like non-standardized function signatures, the need for extensive data processing, and high computational demands. As a result, some upgrades may not have been captured, which could affect the completeness of our analysis. Despite this, we believe that using event-based detection strikes a reasonable balance between accuracy and practicality given the constraints of our study.

Our analysis in RQ3 focuses solely on smart contracts with verified source code, as evaluating certain categories of post-upgrade changes, such as fixing vulnerabilities and feature modifications, relies on access to the source code. Consequently, the findings from this research are primarily applicable to smart contracts where the source code is available.

One potential threat to validity is related to our categorization of post-upgrade changes. Our analysis is based on observable changes in the smart contract code and does not attempt to determine the developer's intention behind each change. While this approach aligns with our research focus on examining the types of changes made in smart contracts from a maintenance perspective, it may not always capture the underlying motivations or root causes of these changes. For example, a change categorized as a security fix could also result in gas optimization, and without considering developer intent, it is challenging to discern if the primary purpose was security or optimization. However, analyzing developer intentions would require extensive manual investigation, including examining documentation or communications, which is beyond the scope of our study due to practical constraints. Therefore, while

our categorization provides insight into the types of maintenance activities performed, there is a potential limitation in not fully capturing the complete context or purpose behind each change.

One limitation of our categorization approach is the handling of minimal changes, particularly those classified as "other." Small modifications that do not clearly align with categories like security, features, or gas optimizations may still be grouped under "other." As a result, the "other" category includes changes that might have broader implications, such as bug fixes or policy-related adjustments, which are not immediately obvious through static analysis. This limitation reflects the inherent difficulty of fully capturing the context and purpose behind every change in smart contracts, particularly for small or ambiguous modifications.

7.4 Conclusion Validity

Our analysis of gas optimizations is based solely on the deployment gas cost, which provides insights into structural changes, such as contract size, storage initialization, and overall complexity. However, this approach does not capture the gas costs of individual function executions, which can vary significantly depending on the function's complexity and usage patterns. As a result, our gas optimization findings may not fully reflect the overall gas efficiency of the contract in real-world scenarios. Additionally, factors like external contract calls, and variations in execution contexts can influence gas usage, which are not accounted for in our analysis. While focusing on deployment gas cost offers a practical means of assessing structural optimizations, future research should include a more granular analysis of function-level gas consumption to provide a more comprehensive understanding of the contract's gas efficiency post-upgrade.

A key limitation in our analysis is the inability to differentiate between transactions made by different proxies reusing the same implementation contract. This could potentially inflate activity levels for certain versions, as a single logic contract may receive calls from multiple proxies, resulting in higher transaction counts. This bias may affect our findings, particularly when assessing the relationship between contract upgrades and activity levels, as the transaction counts could be misattributed to a single version rather than multiple proxies utilizing the same logic contract. Addressing this issue would require access to detailed internal transaction data, which is not readily available through standard blockchain explorers. Full internal transaction analysis demands significant computational resources and a full Ethereum node with tracing capabilities, which exceeds the scope of this study. We acknowledge this limitation and its potential influence on our activity level analysis, which may lead to an overestimation of transaction numbers for certain versions. Further research incorporating internal transaction data could provide a more accurate representation of the activity distribution across proxies and logic contracts.

Some of the interpretations in this study, such as the explanations for the limited prevalence of upgradeable contracts or the shifting trends in proxy standards, are speculative and informed by the lead author's experience, who has over six years of working in the smart contract domain. Additionally, these speculations have been shaped by community discussions observed in forums and developer channels regarding smart contract deployment practices and trends. However, our dataset does not directly capture developer motivations, decision-making processes, or external influences. As such, these interpretations should be viewed as informed hypotheses rather than conclusions supported by direct empirical evidence. We acknowledge this as a limitation and highlight the need for further research or data collection to substantiate these insights.

8 Conclusion

In this paper, we have provided a comprehensive analysis of upgradeable smart contracts, revealing insights into their prevalence, upgrade patterns, and implications. Our study found that upgradeable proxy contracts represent a small fraction of the total smart contracts, indicating a limited adoption of upgradeable patterns. We also observed that the actual frequency of upgrades in smart contracts is comparatively low despite the availability of technical capabilities for frequent modifications. Our analysis of the root causes of upgrades showed diverse motivations, with a significant emphasis on feature modifications and security enhancements, especially when source code is available. In examining the impact of upgradeability on the activity level of contracts, we uncovered a complex relationship between contract versions, their lifespan, and usage, suggesting influences beyond mere technical factors.

For future research, a focused investigation into the security impacts of upgrades, particularly in the context of identified vulnerabilities in various upgrade patterns, would be valuable. This could lead to a better understanding of the security landscape of smart contracts and inform more secure development practices. Furthermore, conducting a survey with smart contract developers and the community could be highly informative. This survey would aim to understand the current issues and challenges developers face in upgrading smart contracts. Insights from such a survey would provide valuable perspectives on the practical difficulties, preferences, and considerations that influence decisions around smart contract upgrades. These findings inform strategies to address these challenges, leading to more efficient and secure approaches to smart contract development.

Data Availability Statements (DAS)

The supplementary material supporting the findings of this study is included in the online supplementary package.[45] Upon acceptance of the paper, the datasets, code, and additional resources utilized in this research will be openly accessible in a designated public GitHub repository.

Conflict of Interest (COI)

The authors declare that they have no conflicts of interest.

References

Albert, E., Correas, J., Gordillo, P., Román-Díez, G., & Rubio, A. (2020). Gasol: Gas analysis and optimization for ethereum smart contracts. In *International conference on tools and algorithms for the construction and analysis of systems* (pp. 118–125).

Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., ... others (2018). Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth eurosys conference* (pp. 1–15).

Antonino, P., Ferreira, J., Sampaio, A., & Roscoe, A. (2022). Specification is law: Safe creation and upgrade of ethereum smart contracts. *arXiv preprint arXiv:2205.07529*.

Araoz, M., & Izquierdo, J. (2018, Feb). *Erc-897: Delegateproxy [draft].* `https://eips.ethereum.org/EIPS/eip-897`.

Atzei, N., Bartoletti, M., & Cimoli, T. (2017). A survey of attacks on ethereum smart contracts (sok). In *Principles of security and trust: 6th international conference, post 2017, held as part of the european joint conferences on theory and practice of software, etaps 2017, uppsala, sweden, april 22-29, 2017, proceedings 6* (pp. 164–186).

Barros, G., & Gallagher, P. (2019, Mar). *Erc-1822: Universal upgradeable proxy standard (uups).* `https://eips.ethereum.org/EIPS/eip-1822`.

Bodell III, W. E., Meisami, S., & Duan, Y. (2023). Proxy hunting: Understanding and characterizing proxy-based upgradeable smart contracts in blockchains. In *32nd usenix security symposium (usenix security 23)* (pp. 1829–1846).

Brown, R. G., Carlyle, J., Grigg, I., & Hearn, M. (2016). Corda: an introduction. *R3 CEV, August*, *1*(15), 14.

Bui, V. C., Wen, S., Yu, J., Xia, X., Haghighi, M. S., & Xiang, Y. (2021). Evaluating upgradable smart contract. In *2021 ieee international conference on blockchain (blockchain)* (pp. 252–256).

[45] `https://drive.google.com/drive/folders/1xlY7suki5TqiOF7mipKspfBbTb888-iz?usp=sharing`

Buterin, V., et al. (2013). Ethereum white paper. *GitHub repository*, *1*, 22–23.

Buterin, V., et al. (2014). A next-generation smart contract and decentralized application platform. *white paper*, *3*(37), 2–1.

Chapin, N., Hale, J. E., Khan, K. M., Ramil, J. F., & Tan, W.-G. (2001). Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice*, *13*(1), 3–30.

Chen, H., Pendleton, M., Njilla, L., & Xu, S. (2020). A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)*, *53*(3), 1–43.

Chen, J. (2020). Finding ethereum smart contracts security issues by comparing history versions. In *Proceedings of the 35th ieee/acm international conference on automated software engineering* (pp. 1382–1384).

Chen, J., Xia, X., Lo, D., Grundy, J., & Yang, X. (2021). Maintenance-related concerns for post-deployed ethereum smart contract development: issues, techniques, and future challenges. *Empirical Software Engineering*, *26*(6), 1–44.

Chen, T., Li, X., Luo, X., & Zhang, X. (2017). Under-optimized smart contracts devour your money. In *2017 ieee 24th international conference on software analysis, evolution and reengineering (saner)* (pp. 442–446).

Clack, C. D., Bakshi, V. A., & Braine, L. (2016). Smart contract templates: Foundations, design landscape and research directions. *arXiv preprint arXiv:1608.00771*, 1-1.

Ebrahimi, A. M., Adams, B., Oliva, G. A., & Hassan, A. E. (n.d.-a). A large-scale exploratory study on the proxy pattern in ethereum.

Ebrahimi, A. M., Adams, B., Oliva, G. A., & Hassan, A. E. (n.d.-b). Upc sentinel: An accurate approach for detecting upgradeability proxy contracts in ethereum.

Ebrahimi, A. M., Adams, B., Oliva, G. A., & Hassan, A. E. (2024). A large-scale exploratory study on the proxy pattern in ethereum. *Empirical Software Engineering*, *29*(4), 1–51.

Frowis, M., & Bohme, R. (2022). Not all code are create2 equal.

He, D., Deng, Z., Zhang, Y., Chan, S., Cheng, Y., & Guizani, N. (2020). Smart contract vulnerability analysis and security audit. *IEEE Network*, *34*(5), 276–282.

Huang, Y., Wu, X., Wang, Q., Qian, Z., Chen, X., Tang, M., & Zheng, Z. (2024). The sword of damocles: Upgradeable smart contract in ethereum. In *Proceedings of the 32nd ieee/acm international conference on program comprehension* (pp. 333–345).

Li, X., Yang, J., Chen, J., Tang, Y., & Gao, X. (2024). Characterizing ethereum upgradable smart contracts and their security implications. In *Proceedings of the acm on web conference 2024* (pp. 1847–1858).

Liu, Y., Li, S., Wu, X., Li, Y., Chen, Z., & Lo, D. (2024). Demystifying the characteristics for smart contract upgrades. *arXiv preprint arXiv:2406.05712*.

Mohanta, B. K., Panda, S. S., & Jena, D. (2018). An overview of smart contract and use cases in blockchain technology. In *2018 9th international conference on computing, communication and networking technologies (icccnt)* (pp. 1–4).

Mudge, N. (2018, Oct). *Erc-1538: Transparent contract standard.* `https://eips.ethereum.org/EIPS/eip-1538`.

(@mudgen), N. M. (2020, Feb). *Erc-2535: Diamonds, multi-facet proxy.* `https://eips.ethereum.org/EIPS/eip-2535`.

Nelaturu, K., Beillahi, S. M., Long, F., & Veneris, A. (2021). Smart contracts refinement for gas optimization. In *2021 3rd conference on blockchain research & applications for innovative networks and services (brains)* (pp. 229–236).

OpenZeppelin. (2018, Apr). *Proxy patterns.* `https://blog.openzeppelin.com/proxy-patterns`. Author.

Palladino, S., Giordano, F., & Croubois, H. (2019, Apr). *Erc-1967: Proxy storage slots.* `https://eips.ethereum.org/EIPS/eip-1967`.

Qian, P., Liu, Z., He, Q., Huang, B., Tian, D., & Wang, X. (2022). Smart contract vulnerability detection technique: A survey. *arXiv preprint arXiv:2209.05872*.

Rodler, M., Li, W., Karame, G. O., & Davi, L. (2021). {EVMPatch}: Timely and automated patching of ethereum smart contracts. In *30th usenix security symposium (usenix security 21)* (pp. 1289–1306).

Salehi, M., Clark, J., & Mannan, M. (2022). Not so immutable: Upgradeability of smart contracts on ethereum. *arXiv preprint arXiv:2206.00716*.

Samreen, N. F., & Alalfi, M. H. (2021). A survey of security vulnerabilities in ethereum smart contracts. *arXiv preprint arXiv:2105.06974*.

Sayeed, S., Marco-Gisbert, H., & Caira, T. (2020). Smart contract: Attacks and protections. *IEEE Access*, *8*, 24416–24427.

Soud, M., Liebel, G., & Hamdaqa, M. (2024). A fly in the ointment: an empirical study on the characteristics of ethereum smart contract code weaknesses. *Empirical Software Engineering*, *29*(1), 13.

Soud, M., Qasse, I., Liebel, G., & Hamdaqa, M. (2023). Automesc: Automatic framework for mining and classifying ethereum smart contract vulnerabilities and their fixes. In *2023 49th euromicro conference on software engineering and advanced applications (seaa)* (pp. 410–417).

Wang, S., Ouyang, L., Yuan, Y., Ni, X., Han, X., & Wang, F.-Y. (2019). Blockchain-enabled smart contracts: Architecture, applications, and future trends. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, *49*(11), 2266–2277.

Wang, S., Yuan, Y., Wang, X., Li, J., Qin, R., & Wang, F.-Y. (2018). An overview of smart contract: architecture, applications, and future trends. In *2018 ieee intelligent vehicles symposium (iv)* (pp. 108–113).

Williams, B. J., & Carver, J. C. (2010). Characterizing software architecture changes: A systematic review. *Information and Software Technology*, *52*(1), 31–51.

Yashavant, C. S., Kumar, S., & Karkare, A. (2022). Scrawld: A dataset of real world ethereum smart contracts labelled with vulnerabilities. *arXiv preprint arXiv:2202.11409*.

Zhang, Z., Zhang, B., Xu, W., & Lin, Z. (2023). Demystifying exploitable bugs in smart contracts. In *2023 ieee/acm 45th international conference on software engineering (icse)* (pp. 615–627).

Chapter 6

Beyond the Upgrade: An Empirical Study of Smart Contract Upgrades and Their Impact on Security

Authors: Ilham Qasse, Mohammad Hamdaqa, and Björn Þór Jónsson
Status: Submitted to ACM Transactions on Software Engineering and Methodology (TOSEM) Journal (Under Review).

Highlights

- Conducts an empirical analysis of smart contract upgrades, assessing the security impact of successive contract versions.
- Identifies a concerning trend where 62% of contract versions fail to resolve or introduce new vulnerabilities during upgrades.
- Introduces a unique dataset tracking security vulnerabilities across smart contract versions, enhancing research resources in this field.
- Uses a standardized taxonomy to classify and analyze vulnerabilities, ensuring consistency and clarity in security evaluation.
- Finds that while more frequent upgrades can resolve some vulnerabilities, they are also linked to the introduction of new issues, particularly gas-related inefficiencies and function visibility flaws.

Beyond the Upgrade: An Empirical Study of Smart Contract Upgrades and Their Impact on Security

ILHAM QASSE, Reykjavik University, Iceland
MOHAMMAD HAMDAQA, Polytechnique Montreal, Canada and Reykjavik University, Iceland
BJÖRN ÞÓR JÓNSSON, Reykjavik University, Iceland

Smart contracts, which are foundational to decentralized applications (DApps), require regular upgrades to maintain functionality, improve security, and adapt to regulatory changes. The security impact of these upgrades remains unclear, however, as they can both resolve and introduce new vulnerabilities. This study empirically investigates how successive smart contract upgrades affect security outcomes. We analyze multiple contract versions to evaluate whether upgrades effectively address or introduce new vulnerabilities. We develop a unique dataset that tracks vulnerabilities across versions using a standardized taxonomy for vulnerabilities and Abstract Syntax Trees (ASTs) to assess changes over time. Our analysis explores five main research questions: the relationship between upgrade frequency and security, the overall impact of upgrades, the most common vulnerabilities resolved and introduced, and those that persist across versions. The results show that although more frequent upgrades are associated with resolving some vulnerabilities, they also correlate with introducing new ones, such as gas-related issues and improper function visibility. A significant portion of contract versions (62%) failed to resolve existing vulnerabilities or introduced new security risks, indicating that current upgrade practices often lack the rigor needed to maintain security. Moreover, certain vulnerabilities, such as gas inefficiencies, outdated compiler versions, and incorrect coding practices, persist across multiple versions, underscoring the need for more comprehensive strategies to manage these issues effectively.

CCS Concepts: • **Security and privacy → Software security engineering**; • **Software and its engineering → Software evolution**; **Maintaining software**.

Additional Key Words and Phrases: Smart contract, Vulnerability, Security, Upgrading, Immutability, Proxy contracts

ACM Reference Format:
Ilham Qasse, Mohammad Hamdaqa, and Björn Þór Jónsson. 2024. Beyond the Upgrade: An Empirical Study of Smart Contract Upgrades and Their Impact on Security. 1, 1 (November 2024), 42 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The emergence of smart contracts represents a transformative step in digitizing contractual agreements [1–4]. These self-executing agreements enable trustless transactions and are foundational to a range of decentralized applications (DApps), extending across various sectors such as finance, real estate, supply chain management, healthcare, and gaming, thereby enhancing the blockchain's adoption [1]. Like any other software system, however, smart contracts are prone to vulnerabilities [5–8], necessitating upgrades to refine functionality, enhance security, and adhere to evolving

Authors' Contact Information: Ilham Qasse, Reykjavik University, Reykjavik, Iceland, ilham20@ru.is; Mohammad Hamdaqa, Polytechnique Montreal, Montreal, Canada, mhamdaqa@polymtl.ca and Reykjavik University, Reykjavik, Iceland, mhamdaqa@ru.is; Björn Þór Jónsson, Reykjavik University, Reykjavik, Iceland, bjorn@ru.is.

Manuscript submitted to ACM

regulations [9, 10]. Nevertheless, upgrade mechanisms must be designed and used with caution; while intended to improve security, they can introduce new avenues for attacks [11]. A notable example of such situation occurred in November 2017 with the Parity multi-signature wallet, where an upgrade attempt to resolve a known vulnerability unexpectedly led to a more critical issue, freezing over $150 million in Ether.[1] This incident illustrates the complexity and potential risks associated with modifying smart contracts and emphasizes the importance of thorough research into the implications of these upgrades.

Despite the prevalence of individual cases and theoretical discussions, there is a significant gap in empirical research focused on understanding the impact of contract upgrades on security outcomes. The existing literature primarily addresses the vulnerabilities present at the initial deployment stage, with limited exploration of how successive upgrades influence the security of these contracts. This paper addresses the crucial yet underexplored topic of how upgrades impact the security of smart contracts, aiming to illuminate the empirical relationship between contract iteration and vulnerability. Given the relevance of this topic, a pressing question arises: Does upgrading smart contracts inherently improve their security, or does it inadvertently compound their vulnerability?

To answer this question, we conducted an empirical study using security detection tools to analyze the vulnerabilities that smart contracts accumulate or resolve through successive upgrades. This analysis required a detailed dataset that tracks smart contract versions and their associated vulnerabilities. However, creating such a dataset presented several challenges. Different security analysis tools often use varying terminologies to describe the same vulnerabilities, complicating the process of tracking and comparing vulnerabilities across versions. To address this, we applied a standardized taxonomy to unify vulnerability names, ensuring consistent labeling across our dataset. Another significant challenge involved analyzing code changes between contract versions. As smart contracts evolve, the location and structure of vulnerabilities can shift, making it difficult to determine whether a vulnerability has been fixed, persists, or has been newly introduced. We addressed this challenge by using Abstract Syntax Trees (ASTs) to compare the code's structural and functional similarities across different versions. This method allowed us to accurately identify and categorize vulnerabilities, even when the code had undergone substantial changes. Using this dataset, we conducted our empirical study to answer key research questions, focusing on the relationship between upgrades and security in smart contracts. Our primary research questions (RQs) are designed to thoroughly investigate the empirical data and develop a comprehensive understanding of this relationship:

- **RQ1: Is there a correlation between the number of versions a smart contract undergoes and the overall security improvement or degradation?** This question examines whether frequent updates improve security or introduce new vulnerabilities. We applied Pearson correlation to study the relationship between the number of contract versions and changes in vulnerability counts. The analysis revealed a moderate positive correlation between the number of versions and the resolution of vulnerabilities. However, more frequent updates also correlate with introducing new vulnerabilities, indicating that while iterative upgrades are beneficial, they must be carefully managed to avoid new risks.
- **RQ2: How do smart contract upgrades impact contract security?** This question assesses whether upgrades effectively fix existing vulnerabilities or introduce new security issues. Understanding this impact is crucial for determining whether the upgrade process is beneficial or needs adjustments to reduce risks. Using AST analysis, we compared the security state of contracts before and after upgrades, examining the types and numbers of vulnerabilities. The study found that 38% of contract versions led to security improvements, while 62% introduced

[1] https://www.zdnet.com/article/parity-shakes-up-wallet-audits-but-funds-remain-frozen/

new vulnerabilities or failed to resolve existing ones. This suggests that upgrades often bring new risks, especially when changes are made to core functionality or new features are added without sufficient testing.

- **RQ3: What are the most common vulnerabilities fixed while upgrading smart contracts?** Identifying the most frequently fixed vulnerabilities during upgrades can help developers focus on critical issues. We categorized and counted fixed vulnerabilities across versions, concentrating on the most common ones. The analysis shows that developers often address vulnerabilities that significantly affect contract functionality and security, such as gas-related issues, permission controls, and function visibility settings. These issues are prioritized because of their direct impact on the contract's efficiency and security. Vulnerabilities like incorrect function signatures and array handling are also consistently resolved, highlighting their importance in maintaining the correct operation of smart contracts.
- **RQ4: What are the most common vulnerabilities introduced while upgrading smart contracts?** This question explores which vulnerabilities are most frequently introduced during upgrades. Recognizing these patterns can help developers and auditors take preventive measures in future versions. By comparing ASTs across versions, we identified and categorized new vulnerabilities introduced during upgrades. The findings show that vulnerabilities most frequently introduced often mirror frequently fixed ones, such as gas-related issues and visibility problems. This suggests that while developers address these issues, introducing new features or changes to existing logic often reintroduces them. Additionally, vulnerabilities like DoS with failed calls and unprotected Ether withdrawal are more likely to emerge in later versions, particularly when new features are added without adequate security measures.
- **RQ5: Do some vulnerabilities persist across versions despite upgrade efforts?** Persistent vulnerabilities are particularly concerning as they indicate that upgrade efforts have not resolved specific issues. Understanding why these vulnerabilities persist can provide insights into the limitations of current upgrade strategies. We tracked vulnerabilities across multiple versions to identify which issues remained unresolved despite upgrades. Several vulnerabilities persist across multiple versions, including gas-related inefficiencies, outdated compiler versions, and incorrect coding practices. These issues often persist due to their complexity, requiring significant refactoring or architectural changes. Their persistence highlights the need for more advanced strategies to resolve these issues fully.

This paper seeks to provide a novel empirical perspective to the conversation on smart contract security, filling the research void with data-driven insights. Hence, the main contributions of this study are twofold: (i) we introduce an extensive dataset of smart contract vulnerabilities, specifically focusing on their evolution, offering a valuable resource for future research and development in this field, and (ii) we provide a detailed empirical analysis of the impact of smart contract upgrades on security, offering insights into the nature and persistence of vulnerabilities.

Key findings reveal that while upgrading smart contracts can resolve existing vulnerabilities, it often introduces new ones and fails to eliminate certain persistent issues completely. While necessary, this indicates that the upgrade process requires more robust strategies to manage and mitigate security risks effectively.

The remainder of the paper is organized as follows. Section 2 provides an overview of smart contracts and their upgradability. Section 3 discusses the proposed research methodology to answer the research questions. We present the results and discussion in Sections 4 and 5, respectively. Future research directions are discussed in Section 6. Section 7 presents the related work. Section 8 discusses threats to research validity. Finally, the paper is concluded in Section 9.

2 Background

This section introduces key concepts relevant to the research questions of this study, including an overview of smart contracts, their upgradeability, and the various approaches and tools used for detecting vulnerabilities.

2.1 Smart Contracts

Smart contracts are self-executing programs that automatically enforce the terms of an agreement when predefined conditions are met, operating on blockchain networks [1–4, 12]. These contracts eliminate the need for intermediaries by providing a decentralized, transparent, and secure way to manage transactions and agreements. Smart contracts are deployed on several blockchain platforms, including Hyperledger Fabric [13], Corda [14], and Ethereum [15]. This study focuses on Ethereum due to its widespread use, robust developer community, and open-source nature.

The primary language for developing Ethereum smart contracts is Solidity [16]. Solidity is a statically typed, object-oriented language designed for writing smart contracts that run on the Ethereum Virtual Machine (EVM). It includes unique features tailored to the needs of smart contract development, such as function visibility modifiers (e.g., 'public,' 'private'), contract-specific operations (e.g., 'self-destruct,' 'revert'), and support for inheritance and libraries. These features make Solidity particularly well-suited for the modular and upgradable nature of smart contracts, which is a key focus of this study. With millions of contracts deployed on Ethereum, Solidity's prominence underscores its importance in the blockchain ecosystem.

2.2 Smart Contract Upgradeability

Upgradeability in smart contracts refers to modifying the contract code after it has been deployed without altering the contract's data and state [9, 10]. This feature is critical for maintaining the longevity and security of smart contracts, as it allows developers to patch vulnerabilities, fix bugs, and introduce new functionalities without disrupting the contract's ongoing operations.

Despite blockchain's inherent immutability, which ensures that once a contract is deployed, it cannot be altered, several design patterns have been developed to achieve upgradeability. These methods work around the immutability constraint by separating the contract's logic from its data, allowing the logic to be updated while preserving it. The primary methods for enabling upgradeability include:

- **Proxy Pattern**: This is the most commonly used approach for achieving upgradeability in Ethereum smart contracts. The proxy pattern involves deploying a proxy contract as an intermediary between users and the logic contract (the contract containing the actual business logic). The proxy contract holds the state of the contract and delegates function calls to the logic contract. When an upgrade is needed, a new logic contract is deployed, and the proxy contract is updated to point to this new contract. This method ensures that the state remains consistent across upgrades. Variants of the proxy pattern, such as the Diamond Proxy and Universal Upgradeable Proxy Standard (UUPS), offer different ways to organize and manage the storage and logic separation, providing flexibility in how upgrades are handled.
- **Data Separation Pattern**: The contract is split into separate storage and logic components in this approach. The storage contract contains the state variables, while the logic contract contains the business logic. The logic contract interacts with the storage contract to read and write data. When an upgrade is required, the logic contract can be replaced with a new version, while the storage contract remains unchanged, ensuring data

continuity. This method is less common than the proxy pattern but is useful in scenarios where strict separation of data and logic is required.

- **Contract Migration**: This method involves deploying a new version of the contract and migrating the state from the old contract to the new one. Users are then encouraged to interact with the new contract. This approach is more manual and less seamless than proxy or data separation patterns, as it requires coordination with users and may involve additional transactions to transfer state. However, it is sometimes necessary for contracts that do not originally include upgradeability mechanisms.

In this study, the focus is on proxy-based upgradeability due to its structured approach, which facilitates automatic identification and versioning. The proxy pattern's distinct and recognizable structure makes it particularly suitable for automated analysis, allowing us to track the evolution of smart contracts across different versions systematically. While other upgrade methods, such as data separation and contract migration, are also viable, they lack the clear, standardized patterns needed for the automated versioning approach used in this research.

2.3 Smart Contract Vulnerability Detection

Detecting vulnerabilities in smart contracts is critical to ensuring their security, particularly given the financial stakes involved in their operation. Several approaches to vulnerability detection are employed in the analysis of smart contracts, each with its strengths and limitations:

- **Static Analysis**: Static analysis involves examining the contract code without executing it. Tools such as Slither [17] and SmartCheck [18] are used to scan the Solidity codebase for known vulnerability patterns, such as reentrancy, integer overflows, and unhandled exceptions. Static analysis is fast and can cover the entire codebase, effectively detecting common vulnerabilities. However, it may miss issues that only arise during runtime or under specific execution conditions.
- **Dynamic Analysis**: Dynamic analysis involves executing the smart contract in a controlled environment to observe its behavior and detect vulnerabilities that manifest during execution. Tools like Mythril[2] and Maian [19] simulate interactions with the contract, identifying issues such as reentrancy attacks, gas limit problems, and scenarios where contracts might become locked or inaccessible. This approach is particularly useful for uncovering vulnerabilities that depend on specific transaction sequences or environmental factors, providing a more realistic assessment of the contract's security.
- **Symbolic Execution**: Symbolic execution combines static and dynamic analysis by exploring all possible execution paths of a contract using symbolic inputs instead of concrete values. Tools such as Oyente [20] and Mythril employ symbolic execution to detect path-specific logical flaws and vulnerabilities, such as conditions that can only be triggered under certain circumstances. Symbolic execution is powerful for identifying complex vulnerabilities but is computationally intensive, making it less scalable for large contracts or extensive testing.
- **Fuzzing**: Fuzzing involves generating many random inputs to test the contract and observe its behavior under unexpected conditions. Tools like sFuzz [21] and ConFuzzius [22] use fuzzing to discover vulnerabilities that emerge from edge cases or non-standard inputs. Fuzzing is effective for finding bugs that are not easily detected through deterministic testing, particularly those related to input validation and error handling.
- **Static Analysis with Formal Methods**: Tools like Securify [23] use static analysis enhanced with formal methods to ensure smart contracts adhere to specific security properties. Securify checks for compliance with a

[2] https://github.com/ConsenSys/mythril

predefined set of security rules, identifying issues such as reentrancy, integer overflows, and unchecked send calls. While it applies formal methods, it is not a full formal verification tool but provides a high level of assurance through rigorous analysis.

Each approach plays a critical role in the comprehensive security assessment of smart contracts. This study employed a combination of these techniques to maximize coverage and ensure thorough vulnerability detection. The selection of specific tools and methods is discussed in detail in Section 3.3.1.

3 Study Method

Figure 1 summarises our approach to smart contract version vulnerability analysis, described in detail in this section, including the data sources (Section 3.1), identification of smart contract versions (Section 3.2), security vulnerability analysis using tools such as Mythril [24], Slither [17], and Smartcheck [18] (Section 3.3), and finally, the vulnerability evolution analysis (Section 3.4) to answer the research questions.

3.1 Data Sources

Our primary data sources are EthereumETL,[3] and Etherscan.[4] These platforms are chosen for their extensive coverage and reliability in providing detailed information about Ethereum smart contracts.

- EthereumETL: This dataset allows us to capture a historical perspective of smart contracts, including their creation, execution transactions, and state changes. This historical view is vital for understanding the lifecycle of smart contracts, particularly for identifying when and how they undergo upgrades. We extract from this dataset proxy smart contracts and the upgrade transactions, which include smart contract version addresses.
- Etherscan: A widely recognized Ethereum block explorer to access detailed information about smart contracts. We use Etherscan to extract source codes for smart contract versions (if available). The collected data is securely stored and efficiently organized for easy access and analysis.

3.2 Smart Contract Versions

This step systematically identifies upgradeable proxy contracts and their corresponding versions within the Ethereum blockchain. This step is integral to assessing the security impacts of smart contract upgrades. The methodology is based on the research presented by Bodell III et al. [25] and Qasse et al. [26]. These studies offer a comprehensive understanding of proxy patterns in smart contracts, providing a solid foundation for our analysis. To identify smart contract versions, we first identified upgradable proxy contracts, and then, based on these contracts, we extracted different smart contract versions. The output of the process described here is a systematically compiled dataset of upgradeable proxy contracts and their respective versions.

3.2.1 Proxy Contract Identification. Identifying proxy contracts is the primary step. These proxies are central to smart contract upgrades, linking old and new versions while maintaining continuity. The process involves detecting the *delegatecall* function, commonly used in proxy contracts. We utilized the *evm-proxy-detection* tool[5] to identify smart contracts that implement recognized proxy patterns efficiently.

[3] https://ethereum-etl.readthedocs.io/en/latest/
[4] https://etherscan.io
[5] https://github.com/gnosis/evm-proxy-detection

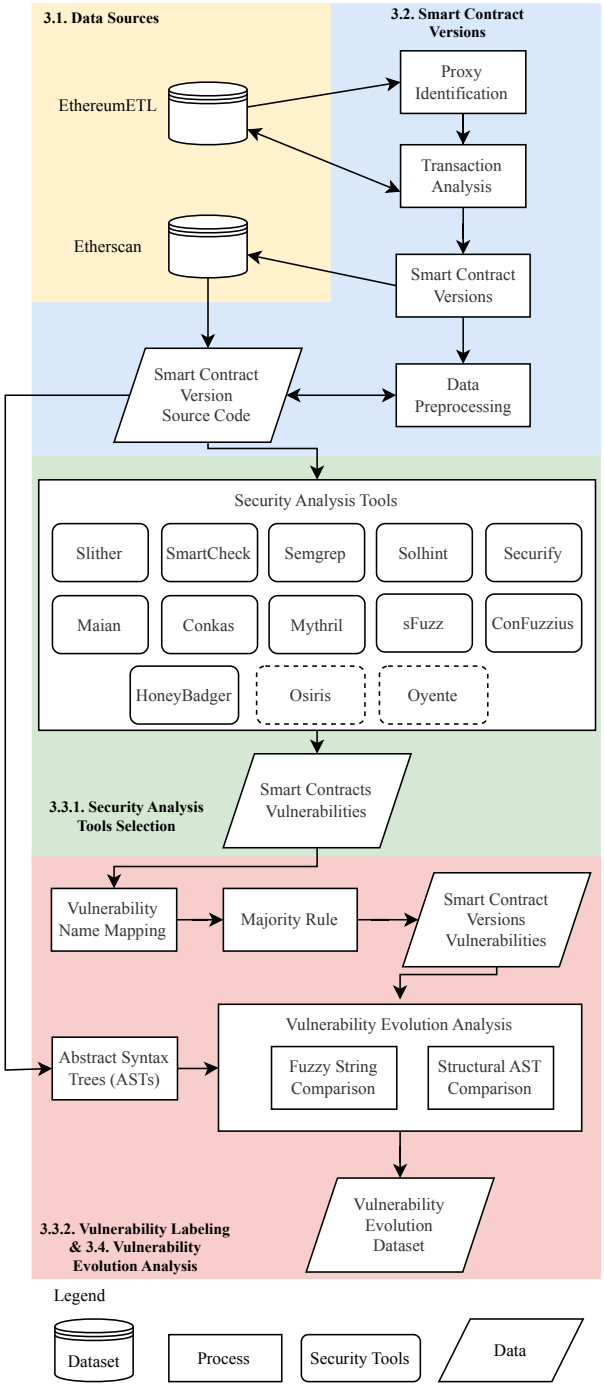

Fig. 1. Overview of the study method. Each coloured box corresponds to a subsection in Section III.

3.2.2 Transaction Analysis. After identifying proxy contracts, the analysis focuses on their transaction histories extracted from the EthereumETL dataset. The study uncovers instances of smart contract upgrades by filtering transactions associated with these proxies, noting the specific addresses of new contract versions.

3.2.3 Source Code Retrieval. Retrieving the source code of various contract versions is essential for the impending security analysis. These source codes are acquired from Etherscan if they are available.

3.2.4 Data Preprocessing. This step aims to refine the collected data, ensuring it is suitable for an in-depth analysis of upgradeable smart contracts. This involves:

- Removal of duplicate contract addresses: Redundant smart contract addresses, which do not contribute to the analysis, are removed from the dataset. This step prevents unnecessary repetition in the dataset, keeping it concise and focused on relevant information.
- Exclusion of Single-Version Proxy Contracts: Proxy contracts with only a single version are excluded. The study requires contracts with multiple versions to compare the changes brought by upgrades. Including single-version contracts would not align with the objective of analyzing version-based differences in smart contracts.
- Source Code Flattening: The source code of the contracts is flattened by combining multiple contract files into a single file. This process is necessary to address dependency issues, especially in contracts comprising multiple files. Flattening simplifies the code structure, facilitating easier analysis of the contracts.

Upon completion of this preprocessing phase, the dataset is transformed into a format that is optimized for analysis.

3.3 Security Analysis

This step is the core of the study, as it identifies security vulnerabilities in the smart contract version, including security analysis tools selection, tools execution, and vulnerability labeling.

3.3.1 Security Analysis Tools Selection. In the security analysis of smart contracts, selecting the right tools is crucial for identifying vulnerabilities effectively. Given the diversity and complexity of potential issues in Solidity-based contracts, we focused on tools that meet specific criteria essential for thorough and reliable analysis. Our criteria for selecting these tools were based on their compatibility with Solidity, ease of use, breadth of vulnerability coverage, performance and reliability, and validation in the literature. The tools we chose needed to be tested and proven effective in previous studies, ensuring that our analysis would be based on reliable and trusted methodologies.

We chose to base our analysis on tools available through the SmartBugs framework [27, 28], a decision grounded in its comprehensive and curated collection of security analysis tools specifically designed for smart contracts. SmartBugs provides a unified platform that simplifies the execution of these tools, making it an ideal choice for our study. The tools within SmartBugs are well-documented, validated in both academic and industry contexts [29–32], and align well with the criteria we established, particularly in supporting Solidity source code analysis.

Within the SmartBugs framework, we specifically focused on tools compatible with Solidity, ensuring that they could effectively analyze smart contracts written in this language. The selected tools are classified into static analysis tools, dynamic analysis tools, fuzzing tools, and symbolic execution tools. In total, 13 tools have been selected for this study, including two tools (Osiris and Oyente) that were partially included. These tools, along with their classifications, are outlined as follows:

(1) **Slither** [17]: a static analysis tool designed to detect a wide range of vulnerabilities in Solidity smart contracts, such as reentrancy, uninitialized storage, and shadowed variables. It provides over 40 built-in vulnerability

detectors and can be extended with custom scripts, making it a versatile tool for comprehensive code analysis. Slither is also capable of identifying code optimization opportunities, which is a significant advantage for improving the efficiency of smart contracts.

(2) **SmartCheck** [18]: a tool that analyzes Solidity code against a comprehensive set of known security vulnerabilities, including issues like reentrancy, integer overflows, and unchecked external calls. It categorizes vulnerabilities by severity (high, medium, low) and generates detailed reports that help developers prioritize their fixes. SmartCheck also checks for adherence to Solidity coding best practices, making it useful for maintaining high code quality and security.

(3) **Semgrep** [33]: a static analysis tool that supports pattern-based detection of security issues in smart contracts. It allows developers to create custom rules tailored to specific coding patterns or security concerns. Semgrep can detect vulnerabilities such as insecure coding practices, unauthorized access, and mismanagement of sensitive data. Its flexibility in rule creation makes it particularly useful in environments where specific, non-standard security checks are required.

(4) **Solhint** [34]: a linter focused on enforcing Solidity coding standards and best practices. It detects security issues related to code quality, such as the use of deprecated functions, inconsistent coding styles, and potential vulnerabilities in code structure. Solhint is particularly valuable in preventing vulnerabilities that arise from poor coding practices, which can be a common source of security flaws in smart contracts.

(5) **Securify** [23]: a static analysis tool that formally verifies smart contracts against a set of security properties. It checks for various vulnerabilities, including reentrancy, integer overflows, and the correct use of Solidity-specific constructs. Securify provides a comprehensive security report that detects vulnerabilities and provides proof of compliance with the specified security properties. Its use of formal methods ensures that the analysis is thorough and precise.

(6) **Maian** [19]: a dynamic analysis tool designed to identify vulnerabilities that can lead to fund locking or contract freeze scenarios, such as unhandled exceptions and improper state changes. It simulates the execution of smart contracts to detect conditions where funds could become locked, or contracts could be rendered unusable. Maian focuses on critical financial vulnerabilities that could have significant consequences if exploited.

(7) **HoneyBadger** [35]: focuses on detecting honeypot vulnerabilities in smart contracts, where the contract is intentionally designed to deceive attackers. It simulates realistic attack scenarios to analyze the contract's behavior under potential exploitation attempts. HoneyBadger's primary goal is to uncover traps set for attackers, ensuring that contracts are secure and do not inadvertently lure attackers into potentially disastrous situations.

(8) **Mythril** [24]: combines static and dynamic analysis with symbolic execution to detect a broad spectrum of vulnerabilities, including integer overflows, underflows, reentrancy attacks, and other logical flaws. Mythril's symbolic execution capability allows it to explore multiple execution paths within a smart contract, making it highly effective at identifying complex vulnerabilities that simpler analysis methods could miss.

(9) **sFuzz** [21]: a fuzzing tool that generates a wide range of random inputs to test the robustness of smart contracts. It is particularly effective at identifying edge cases and vulnerabilities that may not be apparent through other types of analysis, such as unexpected behavior under extreme input conditions. sFuzz's ability to explore a large input space makes it a powerful tool for uncovering hidden vulnerabilities.

(10) **ConFuzzius** [22]: combines static analysis with fuzzing to detect vulnerabilities in smart contracts. It uses static analysis to identify potentially vulnerable code regions and then applies fuzzing techniques to explore these

areas more deeply. This hybrid approach allows ConFuzzius to uncover common and complex vulnerabilities, including those that might only emerge under specific conditions.

(11) **Conkas** [36]: is another hybrid tool that integrates static analysis with fuzzing techniques. It enhances vulnerability detection by first identifying risky code regions through static analysis and then subjecting them to extensive fuzz testing. Conkas is particularly useful for detecting subtle vulnerabilities that might not be obvious through static analysis, such as logic errors or unexpected interactions between contract functions.

While the inclusion of these tools provided comprehensive coverage, certain tools were excluded from our final selection based on our criteria:

- **Manticore** [37]: a symbolic execution tool that was initially considered for its ability to detect deep logical vulnerabilities through thorough path exploration. However, it was excluded due to its time-intensive analysis process, which made it impractical for our study, particularly when analyzing multiple versions of smart contracts. The prolonged execution time would have significantly hindered the efficiency of our analysis.
- **Osiris and Oyente** [20, 38]: these tools were included but with limitations. Osiris and Oyente are known for their symbolic execution capabilities, which are valuable in detecting specific vulnerabilities, such as integer overflows and reentrancy issues. However, they were only used for analyzing older versions of smart contracts. This limitation was due to compatibility issues with newer Solidity versions (0.6.x and above), where both tools often returned errors or failed to complete the analysis. Osiris and Oyente were employed only when they could successfully analyze a contract without encountering these issues to ensure the accuracy and relevance of our findings.

The execution process involves running these tools against each smart contract version collected in this study. This step was essential for identifying a range of vulnerabilities, including syntax mistakes, logical errors, and security flaws.

3.3.2 Vulnerability Labeling. The process of labeling vulnerabilities in our study involves two critical steps: *Vulnerability Name Mapping* and *Application of the Majority Rule*. These steps are necessary to ensure consistency and reliability in the identification and validation of security issues across different tools.

Vulnerability Name Mapping. When conducting a security analysis of smart contracts, the inconsistent terminology used by different tools to describe the same or related vulnerabilities can be challenging. For example, the well-known vulnerability *Reentrancy* is labeled differently depending on the tool: Slither refers to it as *reentrancy-eth*, Oyente labels it as *Re_Entrancy_Vulnerability*, and ConFuzzius uses the term *Reentrancy*. Additionally, some tools may identify vulnerabilities that are not exactly the same but have a similar impact, such as inefficiencies in gas usage. These can be grouped under a broader classification like *Gas-Related Issues*. For instance, Semgrep might label a specific vulnerability as *unnecessary_checked_arithmetic_in_loop*, while Slither identifies a related issue as *calls-loop*. Although these vulnerabilities are not identical, they contribute to the same problem. This variability in naming underscores the need for a systematic approach to unify vulnerability labels across all tools, ensuring consistency in our analysis.

To standardize the vulnerability labeling, we adopted a hybrid taxonomy approach. The primary taxonomy used in this study is the Smart Contract Weakness Classification (SWC) Registry [39], a classification system specifically designed for Ethereum smart contracts. The SWC Registry includes a total of 37 categories covering a wide range of security issues, such as Reentrancy (SWC-107), integer overflows and underflows (SWC-101), and unchecked call return values (SWC-104). These categories are widely recognized and used in both academic research and industry, making the SWC Registry an ideal foundation for our labeling process [40–42].

However, not all vulnerabilities identified by the tools could be directly mapped to the SWC Registry. To address these cases, we supplemented the SWC Registry with the taxonomy proposed by Zhang et al. [43]. This taxonomy was developed through an empirical study of bugs in Ethereum smart contracts, and vulnerabilities were categorized into several groups that are particularly relevant to Ethereum. This secondary taxonomy provides additional granularity and coverage for issues that fall outside the scope of the SWC classifications.

The mapping process was conducted by two independent experts in smart contract security. Each expert independently classified the vulnerabilities detected by the tools into pre-defined categories within our unified taxonomy, following a closed sorting approach. This ensured that all vulnerabilities were consistently mapped according to a set of categories (e.g., SWC codes). In total, the experts mapped 158 vulnerabilities. To quantify the level of agreement between the experts, we calculated Cohen's Kappa, a statistical measure that accounts for agreement occurring by chance. The observed agreement between the experts was 82.61%, indicating that both raters agreed on most classifications. After accounting for the 10.40% agreement expected by chance, the resulting Kappa value of 0.806 signifies substantial agreement between the two experts. The experts engaged in a consensus process to resolve the 22 cases of disagreement. They discussed each mismatch in detail, shared their reasoning for different classifications, and collaboratively worked through the disagreements. This discussion allowed them to reach a final, agreed-upon classification for each disputed vulnerability.

Table 1 shows how vulnerability names from different tools were standardized according to our unified taxonomy. This table illustrates how vulnerabilities are labeled differently across tools but are consistently mapped to the appropriate category in our unified taxonomy. For instance, the vulnerability *Block values as a proxy for time* is identified by Semgrep as *incorrect_use_of_blockhash*, while sFuzz refers to related issues as *Timestamp_Dependency* and *Block_Number_Dependency*. These vulnerabilities are mapped to *SWC-116: Block values as a proxy for time* because they all involve the misuse of time-related or block-specific variables in the smart contract code. Such variables, like block timestamps or block numbers, can be manipulated or are predictable, making them unreliable for security-critical operations like generating random numbers or controlling the contract flow. By mapping these issues to *SWC-116*, we categorize them under a broader, recognized classification that addresses the risks associated with relying on blockchain-specific attributes that malicious actors can exploit.

Application of the Majority Rule. After standardizing the naming of vulnerabilities across different tools, the next step in our methodology is to confirm the validity of these vulnerabilities. Automated tools, particularly in smart contract security, are known for generating high false positive rates [44, 45]. To address this, we applied the majority rule, based on the approach provided by Yashavant et al.[45]. This approach minimizes the inclusion of false positives and ensures that our analysis is based on data with broad tool consensus.

The majority rule requires that a vulnerability be detected by more than 50% of the tools at the same location to be considered valid. This approach filters out false positives and ensures that our analysis focuses on vulnerabilities that multiple tools consistently identify. For example, if a vulnerability like SWC-107: Reentrancy is detected by at least six of the ten tools at the same location, it is confirmed for further analysis. Table 2 provides a sample of how the majority rule was applied, showing which tools detect each vulnerability in Table 1 and the threshold for the majority rule.

After applying the majority rule, the number of confirmed vulnerabilities was reduced from 271,485 to 7,153. The majority rule also slightly reduced the number of unique vulnerability types, from 46 to 45.

Table 1. Mapping of Vulnerability Names to Unified Taxonomy

Unified Vulnerability Name	Tool-Specific Name	Tool
SWC-107: Reentrancy	Reentrancy	ConFuzzius
	Reentrancy	Conkas
	DAOConstantGas	Securify
	balancer_readonly_reentrancy_getpooltokens	Semgrep
	curve_readonly_reentrancy	Semgrep
	erc777_reentrancy	Semgrep
	Reentrancy_bug	Osiris
	Re_Entrancy_Vulnerability	Oyente
	reentrancy-eth	Slither
	reentrancy-no-eth	Slither
SWC-101: Integer Overflow and Underflow	Integer_Overflow	ConFuzzius
	Integer_Underflow	Conkas
	Integer_Overflow	Conkas
	basic_arithmetic_underflow	Semgrep
	Overflow_bugs	Osiris
	Underflow_bugs	Osiris
	Integer_Overflow	Oyente
	integer arithmetic bugs	Mythril
SWC-116: Block values as a proxy for time	Time_Manipulation	Conkas
	Timestamp_Dependency	sFuzz
	Block_Number_Dependency	sFuzz
	incorrect_use_of_blockhash	Semgrep
	dependence on predictable environment variable (swc 116)	Mythril
SWC-106: Unprotected SELFDESTRUCT Instruction	Unprotected_Selfdestruct	ConFuzzius
	Not_destructible_no_self_destruct	Maian
	unprotected selfdestruct (swc 106)	Mythril
Gas-Related Issues	unnecessary_checked_arithmetic_in_loop	Semgrep
	use_short_revert_string	Semgrep
	calls-loop	Slither
	max_states_count	Solhint-3.3.8
	SOLIDITY_EXTRA_GAS_IN_LOOPS	SmartCheck
	SOLIDITY_GAS_LIMIT_IN_LOOPS	SmartCheck

Table 2. Majority Rule Application Across Tools for Selected Vulnerabilities

Unified Vulnerability Name	Detection Count	Tools	Threshold
SWC-107: Reentrancy	7	ConFuzzius, Conkas, Securify, Semgrep, Osiris, Oyente, Slither	4
SWC-101: Integer Overflow and Underflow	6	ConFuzzius, Conkas, Semgrep, Osiris, Oyente, Mythril	4
SWC-116: Block values as a proxy for time	4	Conkas, sFuzz, Semgrep, Mythril	2
SWC-106: Unprotected SELFDESTRUCT Instruction	3	ConFuzzius, Maian, Mythril	2
Gas-Related Issues	4	Semgrep, Slither, Solhint, SmartCheck	3

3.4 Vulnerability Evolution Analysis

This final stage of smart contract analysis focuses on comparing vulnerabilities across different contract versions to determine their status, whether they have been resolved, persisted, or newly emerged. This comparison provides a detailed map of vulnerabilities associated with each version, which is critical for addressing the research questions. Specifically, the analysis evaluates the status of a vulnerability based on the function where it is identified, classifying vulnerabilities as fixed, persistent, or introduced.

Direct line-by-line comparison of smart contract code between versions is insufficient due to the nature of software evolution. Changes such as code refactoring, function reordering, or variable renaming can alter the appearance of the code without changing its underlying logic. These superficial changes can mislead direct comparisons, leading to false conclusions about whether a vulnerability has been fixed or persists. Therefore, a more sophisticated approach is necessary.

To address the limitations of direct comparison, we use Abstract Syntax Trees (ASTs) to represent the code's structure. An AST provides a hierarchical representation of the code, capturing its syntactic and logical organization. By comparing the ASTs of different versions, we can accurately determine whether the same code (and thus the same vulnerability) persists across versions, regardless of superficial changes.

The analysis begins by parsing the Solidity code of the original and updated versions to generate their respective ASTs. The AST represents the hierarchical structure of the contract's code, capturing functions, control flows, expressions, and other syntactic elements. The `solidity_parser` library is used to generate these ASTs, providing a detailed tree structure of the contract.

After generating the ASTs, the next step is identifying the function where the vulnerability is located in the original contract version. This function and its corresponding AST node are then identified in the subsequent version of the contract. The comparison is performed at the function level, ensuring we account for changes that might affect the entire function rather than individual lines of code.

To formalize this process, we present Algorithm 1, which outlines the steps involved in the vulnerability evolution analysis. The algorithm takes as input the solidity code of two consecutive versions of a contract, the detected vulnerabilities, and the line numbers associated with these vulnerabilities.

To verify whether the function in the current version is equivalent to the one in the previous version, we use two comparison methods within the algorithm:

- **Fuzzy String Comparison**: This method compares textual elements within the AST nodes, such as function names, variable names, and literals. It is designed to tolerate minor changes, such as renaming variables or functions, without treating them as significant alterations. The comparison uses a threshold of 0.5, meaning that if the similarity score is above this threshold, the two elements are considered similar. This threshold was chosen based on manual testing of various examples to balance flexibility and precision. We experimented with multiple cases where minor changes (such as renaming variables or small adjustments to logic) occurred but the overall structure and functionality remained consistent. A higher threshold (e.g., 0.7) proved too strict, often missing legitimate similarities, while a lower threshold (e.g., 0.3) allowed too much flexibility, leading to many false positives. Through manual analysis, we found 0.5 to be optimal, allowing for the detection of meaningful similarities while avoiding the majority of misclassifications. While this threshold works well for most cases, we acknowledge that in certain rare cases, distinct functions with shared patterns may still exceed this threshold.

Algorithm 1 Vulnerability Evolution Analysis Algorithm

Require: *PreviousVersion, CurrentVersion, DetectedVulnerabilities, LineNumbers*
Ensure: *VulnerabilityStatusReport*

```
PreviousAST ← GenerateAST(PreviousVersion)
CurrentAST ← GenerateAST(CurrentVersion)
for each V in DetectedVulnerabilities do
    PreviousFunctionAST ← FindFunctionASTNode(PreviousAST, V.FunctionName, V.LineNumber)
    CurrentFunctionAST ← FindFunctionASTNode(CurrentAST, V.FunctionName)
    if CurrentFunctionAST = None then
        VulnerabilityStatus ← "Fixed"
    else
        SameLocation ← CompareASTNodes(PreviousFunctionAST, CurrentFunctionAST)
        if SameLocation then
            if V in ExtractVulnerabilities(CurrentFunctionAST) then
                VulnerabilityStatus ← "Persistent (Not Fixed)"
            else
                VulnerabilityStatus ← "Fixed"
            end if
        else
            if V in ExtractVulnerabilities(CurrentFunctionAST) then
                VulnerabilityStatus ← "Reintroduced at New Location"
            else
                VulnerabilityStatus ← "Fixed"
            end if
        end if
    end if
    if VulnerabilityStatus = "Fixed" and V not in DetectedVulnerabilities of PreviousVersionAST then
        VulnerabilityStatus ← "Introduced"
    end if
    VulnerabilityStatusReport.append(V, VulnerabilityStatus)
end for
return VulnerabilityStatusReport
```

- **Structural AST Comparison**: This method evaluates the overall structure of the AST nodes, including the function and its children (e.g., control structures, expressions). The structural comparison checks whether the logical organization of the function has changed. If the structure remains similar (similarity score above 0.5), it suggests that the function's logic is largely unchanged, even if some elements have been altered.

The combination of fuzzy string and structural AST comparisons, as outlined in the algorithm, is essential to accurately assess whether a function (and its associated vulnerability) has changed. Fuzzy string comparison handles minor textual differences, while structural comparison ensures that the overall logic and flow of the function are consistent. Using both methods together reduces the risk of incorrectly classifying a vulnerability due to superficial changes, providing a more reliable assessment.

Based on the AST comparisons performed in the algorithm, vulnerabilities are classified into three categories:

(1) **Fixed**: If the function associated with the vulnerability is still present and structurally similar (similarity above 0.5) in the current version, but the vulnerability is no longer present, it is classified as fixed. Additionally, if the

Table 3. Summary of Dataset Statistics

Statistic	Value
Number of Proxies	1,242
Number of Contract Versions	2,746
Maximum Number of Versions per Proxy	11
Mean Number of Versions per Proxy	2.21
Number of Vulnerabilities	7,153
Number of Unique Vulnerabilities	45

function is either absent in the current version or significantly altered (similarity below 0.5), it is also classified as fixed since the vulnerability can no longer be found in its original context.

(2) **Persistent (Not Fixed)**: If the function remains structurally similar (similarity above 0.5) and the vulnerability still exists in the current version, it is classified as persistent, indicating that the issue has not been resolved.

(3) **Introduced**: This category includes newly introduced and reintroduced vulnerabilities. If a new vulnerability that was not present in the previous version appears in the current version, it is classified as introduced. Reintroduced vulnerabilities occur when a vulnerability that was previously identified in an earlier version reappears in a different location in the subsequent version. This happens when the vulnerability is detected in a part of the code that is structurally different from where it was originally found. Although the code may have changed, such as by refactoring or introducing new functionality, the underlying issue was not fully resolved, leading to the vulnerability reappearing in a new location within the code.

The result of this step enables the detection of patterns and trends in vulnerability evolution across versions, offering insights into the security implications of contract upgrades. The resulting dataset is publicly available in the project repository.[6]

4 Results

This section presents the results of our empirical study. We first present the dataset resulting from the process outlined above, before discussing the four research questions and outlining our findings.

4.1 Dataset Overview

The dataset presents an analysis of smart contract proxies, smart contract versions, and their associated vulnerabilities. It contains records from **1,242 unique proxies** and **2,746 unique contract versions** associated with them. The dataset also shows that the highest number of versions associated with a single proxy is **11**, indicating the range of variations individual contracts can undergo. On average, each proxy has **2.21 versions**, underscoring the dynamic nature of smart contract development, where most proxies experience multiple versions and upgrades. Table 3 summarizes the key statistics of the dataset.

Regarding security, the dataset documents **7,153 instances of vulnerabilities** distributed across **45 distinct types**. Figure 2, a bar chart, highlights the five most frequent vulnerabilities. The x-axis represents the types of vulnerabilities, and the y-axis indicates the number of occurrences for each type. It visually captures the prevalence of different vulnerabilities in the dataset, helping to identify the most common security issues encountered in smart contracts.

[6] https://drive.google.com/drive/folders/1DeEewXasRAF2EF0OVtAN7ilHkv9ZxhJD?usp=sharing

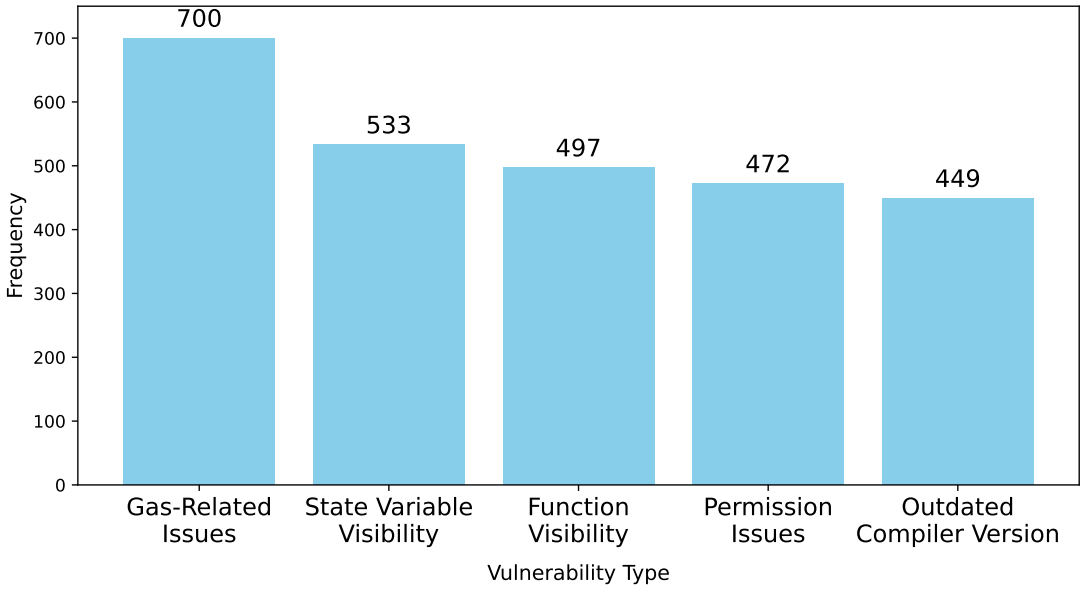

Fig. 2. Top prevalent vulnerabilities in the dataset

- **Gas-Related Issues:** Gas-related issues are the most frequently observed vulnerabilities, appearing in 700 instances. These issues often arise from excessive gas consumption in loops, recursive functions, or poorly optimized code structures. If left unresolved, gas-related issues can render a contract prohibitively expensive or cause transactions to fail due to running out of gas. This directly impacts smart contracts' usability and economic viability, especially on platforms like Ethereum, where gas costs are significant. For example, a loop processing a large array could lead to high gas usage, as shown in Listing 1.

Listing 1. Inefficient loop causing high gas usage

```
// Inefficient loop causing high gas usage
for (uint i = 0; i < largeArray.length; i++) {
    // Some gas-intensive computation
}
```

The loop iterates over a large array in this example, performing a gas-intensive computation. If the array size is large, the gas cost can exceed the block gas limit, causing the transaction to fail. Optimizing such loops or breaking down the computation into smaller transactions can mitigate this risk.

- **SWC-108: State Variable Default Visibility:** This vulnerability, identified 533 times, occurs when state variables are not explicitly marked with visibility modifiers, potentially leading to unintended access or exposure. Such issues can result in unauthorized access to internal contract logic, posing significant security risks. For instance, a state variable intended to be private might inadvertently be accessible publicly, as shown in Listing 2.

Listing 2. State variable with undefined visibility

```
// Vulnerable code
uint balance; // Should be private or internal
```

In this example, the *balance* variable is not explicitly marked as *private* or *internal*. This omission could allow external contracts or users to access and potentially manipulate the *balance* variable, leading to unauthorized withdrawals or state changes.

- **SWC-100: Function Default Visibility:** Found in 497 instances, this vulnerability occurs when functions lack explicit visibility modifiers, leading to potential exposure to external calls. This can allow attackers to invoke internal functions, bypassing intended access controls. For example, a function modifies sensitive contract data might be accessible to anyone, as demonstrated in Listing 3.

Listing 3. Function with default visibility

```
// Vulnerable code
function updateBalance() {
    // Function logic
}
```

In this example, the function *updateBalance* lacks an explicit visibility modifier, making it *public* by default. This allows anyone to call this function, potentially leading to unauthorized changes to the contract's state, which malicious users could exploit.

- **Permission Issues:** Permission issues, identified in 472 instances, arise from weak or incorrectly implemented access control mechanisms. This can lead to unauthorized users performing privileged actions, resulting in significant security breaches. For instance, an insufficiently protected withdraw function might allow any user to drain funds, as shown in Listing 4.

Listing 4. Incorrect permission check

```
// Vulnerable code
function withdraw() public {
    // Incorrect permission check
}
```

In this example, the *withdraw* function is marked as *public* without proper access control checks. This vulnerability could enable any user to call the function and withdraw funds from the contract, leading to potential financial losses.

- **SWC-102: Outdated Compiler Version:** Using outdated compiler versions was observed 449 times. For example, Solidity version 0.4.24, while functional, is outdated and lacks several security features introduced in later versions, such as stricter rules on visibility modifiers, safer arithmetic operations (to prevent overflows and underflows), and better support for static analysis tools. Contracts compiled with this version are more vulnerable to known exploits that have been mitigated in subsequent versions, such as 0.5.x and beyond, which enforce better coding practices and include numerous bug fixes. An example of a pragma statement using this outdated version is shown in Listing 5.

Listing 5. Outdated Solidity version

```
pragma solidity ^0.4.24; // Outdated version
```

This example demonstrates using Solidity version 0.4.24, which lacks modern security features. Contracts using this version are more susceptible to issues like integer overflows, underflows, and other vulnerabilities that have been addressed in later versions. Developers are advised to upgrade to newer versions to benefit from improved security measures.

4.2 RQ1: Correlation Between Smart Contract Version Updates and Security Performance

To explore the relationship between the number of versions a smart contract undergoes and its overall security performance, we employed the Pearson correlation coefficient. The Pearson correlation coefficient is a widely used statistical measure that quantifies the strength and direction of a linear relationship between two variables. The coefficient, denoted by r, ranges from -1 to +1. A value of $r = +1$ indicates a perfect positive linear relationship, where increases in one variable correspond to increases in the other. Conversely, $r = -1$ signifies a perfect negative linear relationship, where increases in one variable correspond to decreases in the other. A value of $r = 0$ suggests no linear relationship between the variables. The strength of the correlation is generally interpreted as follows: r values between 0.00 and 0.29 indicate a weak correlation, r values between 0.30 and 0.49 indicate a moderate correlation, and r values between 0.50 and 1.00 indicate a strong correlation.

The formula for the Pearson correlation coefficient is:

$$r = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2 \sum (y_i - \bar{y})^2}} \tag{1}$$

In this equation, x_i and y_i are individual data points for the variables x and y, respectively. The terms $\bar{x}$ and $\bar{y}$ represent the mean values of x and y. The numerator, $\sum (x_i - \bar{x})(y_i - \bar{y})$, calculates the covariance between the variables, which measures how much the two variables change together. The denominator, $\sqrt{\sum (x_i - \bar{x})^2 \sum (y_i - \bar{y})^2}$, normalizes this covariance by the product of the standard deviations of x and y, thereby scaling the coefficient to lie within the range of -1 to +1.

In the context of this study, we defined the variables x and y as follows:

- x: The total number of versions each smart contract has undergone.
- y_1: The total number of fixed vulnerabilities for each smart contract.
- y_2: The total number of newly introduced vulnerabilities for each smart contract.

We computed three separate correlations:

- The correlation between the total number of versions (x) and the total number of fixed vulnerabilities (y_1).
- The correlation between the total number of versions (x) and the total number of newly introduced vulnerabilities (y_2).
- The correlation between the total number of fixed vulnerabilities (y_1) and the total number of newly introduced vulnerabilities (y_2).

The results of these calculations are summarized in the correlation matrix shown in Figure 3, which provides a comprehensive overview of the relationships between the variables.

The correlation between the number of versions and the number of fixed vulnerabilities is $r = 0.41$, indicating a moderate positive relationship. This suggests that as the number of versions increases, there is a noticeable trend

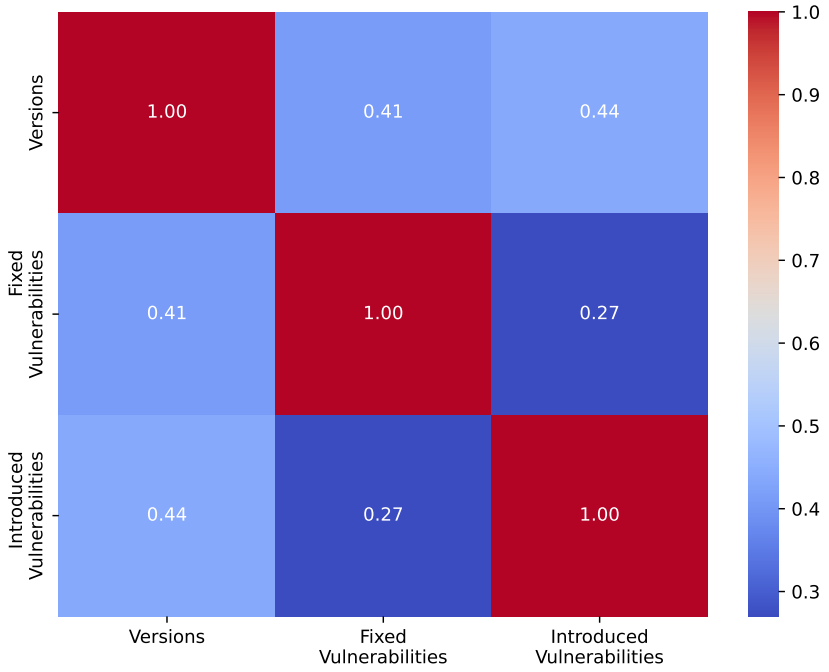

Fig. 3. Correlation Matrix: Smart Contract Versions and Vulnerabilities (RQ1)

towards resolving existing vulnerabilities. However, the correlation is moderate rather than strong, implying that while version updates contribute to fixing vulnerabilities, other factors also influence the effectiveness of these fixes.

Similarly, the correlation between the number of versions and newly introduced vulnerabilities is $r = 0.44$, which is slightly stronger than the correlation with fixed vulnerabilities. This finding suggests that while updates are intended to improve security, they are also associated with introducing new vulnerabilities. The slightly stronger correlation here may indicate that updating and adding new features or changes tends to introduce new security challenges, perhaps more so than it contributes to fixing existing issues. This relationship underscores the complexity of maintaining smart contract security over multiple versions, where the benefits of fixing vulnerabilities must be balanced against the risks of introducing new ones.

The correlation between fixed and introduced vulnerabilities is $r = 0.27$, the weakest among the three correlations. This lower correlation suggests that fixing and introducing new vulnerabilities are relatively independent. In other words, the number of vulnerabilities fixed does not strongly predict the number of new vulnerabilities introduced, implying that different factors might influence these two aspects of security management.

RQ1. Is there a correlation between the number of versions a smart contract undergoes and the overall security improvement or degradation?

Answer: The results demonstrated a moderate correlation between the number of versions a smart contract undergoes and its security improvements and potential degradation. More versions are associated with more fixed vulnerabilities, suggesting that updates contribute positively to security improvements. However, the slightly stronger correlation between versions and newly introduced vulnerabilities highlights the inherent risk of introducing new vulnerabilities with each update. This suggests that while versioning is necessary for maintaining and enhancing security, it must be approached cautiously to balance the benefits of fixing vulnerabilities against the risks of introducing new ones. The relatively weak correlation between fixed and introduced vulnerabilities further emphasizes that these processes are not strongly linked, highlighting the importance of rigorous testing and security measures during the update process.

4.3 RQ2: Impact of Smart Contract Upgrades on Security Vulnerability Evolution

This question refines the focus on the specific impact of upgrades on the evolution of vulnerabilities, both in addressing existing issues (fixes) and introducing new vulnerabilities. It directly ties the process of upgrading smart contracts to the changes in the security landscape.

The analysis of smart contract upgrades reveals two primary categories based on their impact on security: **Positive Impact** and **Negative Impact**. The positive impact category includes versions where upgrades either improved security by fixing vulnerabilities or maintained security without introducing new issues. Conversely, the negative impact category comprises versions where upgrades either failed to address existing vulnerabilities or introduced new ones, thereby compromising the security of the contract.

Figure 4 presents a visual representation of the distribution of these impacts. The x-axis categorizes the impact of smart contract upgrades into two groups: Positive Impact and Negative Impact. The y-axis represents the percentage of smart contract versions within each category. The figure uses a stacked bar chart to illustrate the breakdown within each impact category. For Positive Impact, the chart differentiates between "Fixed Vulnerabilities" and "Good Maintenance," while for Negative Impact, it separates "Introduced Vulnerabilities" and "Unfixed Vulnerabilities."

Positive Impact on Security. The positive impact category accounts for approximately 38% of the analyzed smart contract versions, reflecting upgrades' effectiveness in enhancing or maintaining security. This category is further divided into two subcategories:

- **Good Maintenance (17.83%):** This subcategory includes contracts that successfully preserved their security post-upgrade, with no new vulnerabilities introduced and no existing vulnerabilities needing fixes. This outcome suggests that these contracts were well-designed and could withstand changes without compromising security. The stability observed in this subcategory is a positive indication of effective initial design and cautious upgrade practices. Contracts in this category serve as a model for maintaining security across versions, emphasizing the importance of rigorous testing and conservative deployment practices.
- **Fixed Vulnerabilities (20.47%):** This subcategory demonstrates the success of upgrades in resolving known security issues. Contracts in this group benefited from upgrades that actively enhanced their security posture by addressing vulnerabilities. This proactive approach is essential for minimizing the risk of exploitation and ensuring the long-term reliability of smart contracts. The relatively high percentage of fixed vulnerabilities

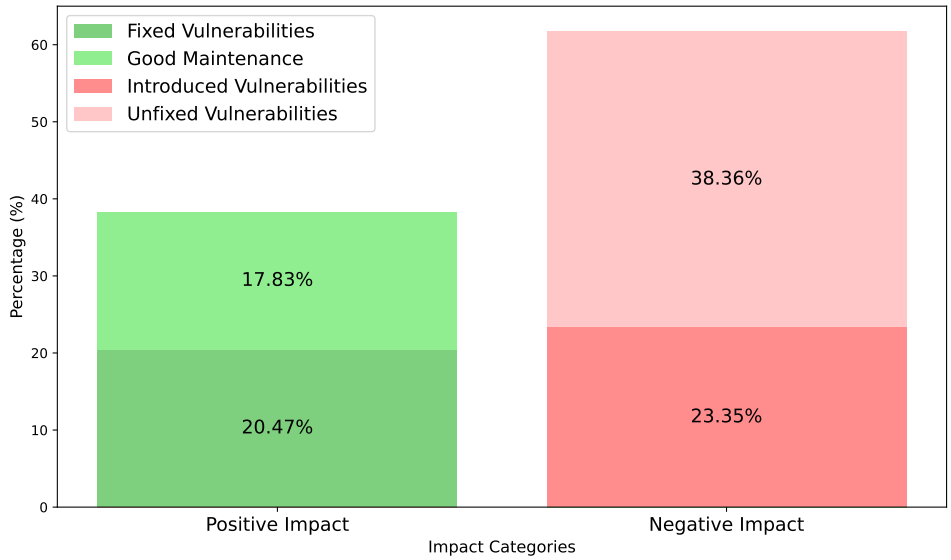

Fig. 4. Impact of smart contract upgrades on security vulnerability evolution (RQ2)

highlights the importance of continuous security improvements and the adoption of best practices in smart contract development. This subcategory underscores the potential of upgrades as a tool for security enhancement when executed with a focus on resolving known existing issues.

Negative Impact on Security. The negative impact category, which accounts for approximately 62% of the analyzed versions, is concerning as it reflects the instances where upgrades either failed to resolve existing vulnerabilities or introduced new ones, thereby compromising the security of the contracts. This category is also divided into two subcategories:

- **Unfixed Vulnerabilities (38.36%):** This subcategory suggests that a significant number of upgrades were insufficient in addressing known security issues. The persistence of these vulnerabilities indicates that developers may prioritize functionality or performance enhancements over the critical need to resolve security flaws. This oversight poses ongoing risks for exploitation, potentially leading to severe consequences such as financial loss or reputational damage. The high percentage of unresolved vulnerabilities calls for a more focused approach to vulnerability management during the upgrade process. Addressing these vulnerabilities should be a priority to reduce the overall security risks associated with smart contracts.
- **Introduced Vulnerabilities (23.35%):** In this subcategory, the upgrade process introduced new security vulnerabilities. This outcome highlights the inherent risks of modifying smart contracts. The introduction of new vulnerabilities suggests that the changes made during upgrades were not adequately vetted for security implications. This finding underscores the necessity for rigorous testing and validation processes to prevent new vulnerabilities from emerging during upgrades. The presence of introduced vulnerabilities reflects the complexity of ensuring security during the upgrade process and the potential for unintended consequences when making changes to contract code.

RQ2. How do smart contract upgrades impact contract security?

Answer: The analysis reveals a dual impact of smart contract upgrades on security. On the positive side, a significant portion of upgrades successfully enhanced security by maintaining stability or fixing existing vulnerabilities. This outcome demonstrates that when upgrades are executed with a strong focus on security, they can effectively mitigate risks and reinforce the reliability of smart contracts. However, the analysis also highlights that most upgrades had a negative impact, either by failing to resolve known vulnerabilities or by introducing new ones. This dual outcome underscores the inherent challenges and risks in the upgrade process. The persistence of unresolved vulnerabilities suggests that developers may not prioritize security adequately during upgrades. Moreover, the introduction of new vulnerabilities points to gaps in testing and validation procedures. Developers must implement more rigorous security testing and vulnerability management practices to address these issues. Prioritizing comprehensive security assessments during the upgrade process is crucial to minimizing negative impacts on security.

4.4 RQ3: Analysis of Prevalent Vulnerabilities Fixed in Smart Contract Upgrades

In smart contract upgrades, certain vulnerabilities are addressed more frequently due to their critical impact on security and functionality, while others are fixed more often relative to their occurrence. Based on the results from RQ2, we focused on the vulnerabilities fixed(20.47%) to understand how developers prioritize different issues. The total number of fixed vulnerabilities amounted to **2,629**, encompassing **43** unique types. This section discusses both the most commonly fixed vulnerabilities and those with more than a 50% fix-to-find ratio, providing insights into why developers prioritize these issues.

Top Fixed Vulnerabilities. Among the most fixed vulnerabilities, certain types were addressed more frequently due to their impact on the security and functionality of smart contracts. Figure 5 shows a bar chart depicting these frequently addressed vulnerabilities. The x-axis displays the different types of vulnerabilities, while the y-axis indicates the total number of fixes for each vulnerability. The top fixed vulnerabilities are:

- **Gas-Related Issues**: Gas-related issues were the most prevalent vulnerabilities and were also the most frequently fixed, with **243** instances addressed out of **700** occurrences, resulting in a fix-to-find ratio of approximately **34.71%**. The high prevalence of gas-related issues is tied to their direct impact on the cost-effectiveness and functionality of smart contracts, as excessive gas consumption can lead to prohibitively high transaction costs or even failures due to running out of gas. Gas-related issues are frequently fixed because of their relatively straightforward detection and remediation. Static analysis tools can easily identify loops or recursive functions with high gas consumption, and developers can often implement optimizations without significant architectural changes. For instance, in the inefficient loop discussed in Listing 1, the fix involves breaking down the loop into smaller, more manageable batches to reduce gas usage, as shown in Listing 6. This fix is relatively simple to implement and immediately reduces gas costs, making it a high-priority fix during upgrades. Developers recognize that optimizing gas usage is essential for ensuring contracts remain practical and cost-efficient, which is why these issues are frequently fixed.

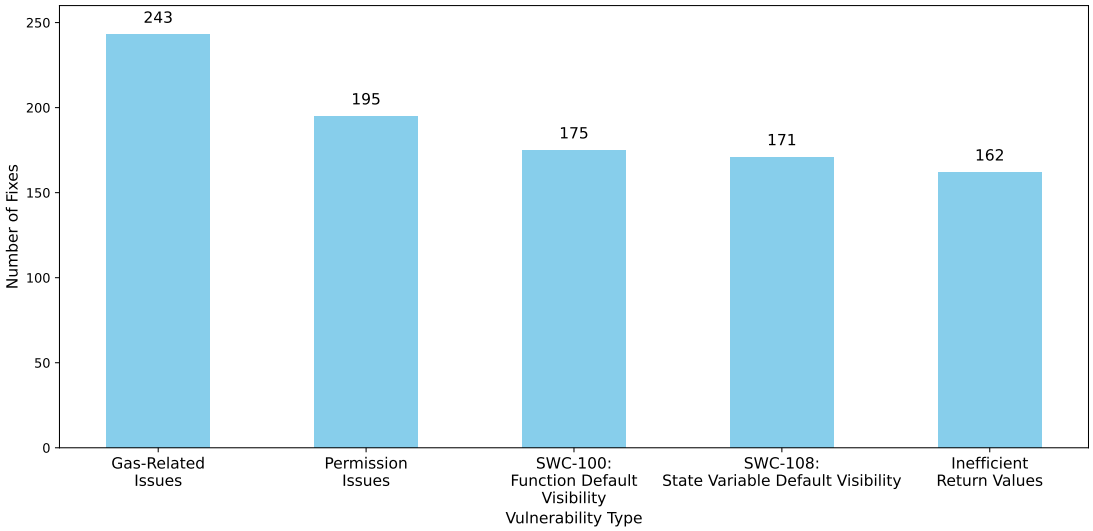

Fig. 5. Top common vulnerabilities fixed in smart contract upgrades (RQ3)

Listing 6. Optimized approach to reduce gas usage

```
// Optimized approach to reduce gas usage
function processBatch(uint start, uint end) public {
    require(end > start && end <= largeArray.length);
    for (uint i = start; i < end; i++) {
        // More efficient computation in smaller chunks
    }
}
```

- **Permission Issues**: Permission issues were the second most common vulnerability, with **195** instances fixed out of **472** occurrences, yielding a fix-to-find ratio of **41.31%**. These vulnerabilities often arise from insufficient access controls, allowing unauthorized users to perform sensitive actions such as transferring funds or modifying contract states.

  The relative ease of detecting and fixing permission issues contributes to their high fix rate. For example, in Listing 4, a `withdraw` function lacked proper access control. The fix, which involved implementing role-based access controls like the `onlyOwner` modifier, is straightforward, as shown in Listing 7. This simple yet effective fix ensures that only authorized users can execute critical operations, significantly enhancing the security of the contract. The frequency with which permission issues are fixed reflects developers' understanding that unauthorized access is a critical threat to contract security and must be addressed promptly.

Listing 7. Implementing access control for ownership

```
address private owner;

modifier onlyOwner() {
```

```
        require(msg.sender == owner, "Caller is not the owner");
        _;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        require(newOwner != address(0), "New owner is the zero address");
        owner = newOwner;
    }
```

- **SWC-100: Function Default Visibility**: Function visibility issues, particularly those classified under SWC-100, were addressed in **175** out of **497** instances, leading to a fix-to-find ratio of **35.21%**. In Solidity, functions default to `public` visibility if not explicitly specified, which can unintentionally expose internal contract functions to external users.
  The ease with which this issue can be detected and corrected likely contributes to its high fix rate. Static analysis tools can flag functions without visibility modifiers, and the fix(adding an explicit visibility modifier)is simple. For example, Listing 3 demonstrated a function that was incorrectly exposed due to the lack of an explicit visibility modifier. Listing 8 shows a possible fix for this vulnerability: setting the function's visibility explicitly. By ensuring that functions are only accessible where necessary, developers can protect the contract from unauthorized interactions. However, it might have been intentional in cases where developers did not fix this issue. Depending on the contract's intended functionality, developers might have wanted the function to be public. Therefore, while adding visibility is a best practice, the absence of a fix does not necessarily indicate a vulnerability in those cases.

Listing 8. Correcting function visibility

```
// Corrected with explicit visibility
function updateData(uint data) private {
    storedData = data;
}
```

- **SWC-108: State Variable Default Visibility**: Issues related to the visibility of state variables, classified under SWC-108, were fixed in **171** out of **533** instances, resulting in a fix-to-find ratio of **32.08%**. Like function visibility, state variables that are not explicitly marked with visibility modifiers can be unintentionally exposed, potentially leading to security risks.
  The introduction of this vulnerability often occurs due to oversight or misunderstanding of Solidity's default visibility rules. However, the fix(adding an explicit visibility modifier)is straightforward and effective. Listing 2 previously showed a state variable that was not properly protected. The fix included specifically defining the visibility of the variable, as shown in Listing 9.This adjustment ensures that the variable is only accessible within the intended scope, preventing unauthorized access. As with SWC-100, in cases where this issue was not fixed, the developer might have intended the variable to be public. Therefore, not every instance of this vulnerability being unfixed is necessarily problematic.

Listing 9. Correcting state variable visibility

```
// Corrected visibility to protect the variable
```

```
    uint256 private publicBalance;
```

- **Inefficient Return Values:** This vulnerability was fixed in **162** out of **404** occurrences, resulting in a fix-to-find ratio of **40.10%**. It occurs when a function in a smart contract returns more data than necessary or fails to optimize the data it returns. This inefficiency leads to higher gas costs and can expose unnecessary data to the public, potentially leading to security risks.
  For instance, Listing 10 demonstrates a scenario where the `getDetails` function returns three pieces of data: `id`, `balance`, and `name`. If the `id` is not needed for most operations, its inclusion results in unnecessary gas costs each time the function is called. The optimized version, shown in the same listing, reduces this inefficiency by returning only the `balance`, thereby minimizing gas costs and exposure of unnecessary data. The optimized function section in Listing 10 shows that the fix, which involves reducing the amount of data returned, is straightforward to implement. This change provides immediate cost reduction and security benefits.

Listing 10. Inefficient function returning unnecessary data and optimized version

```
// Inefficient function returning unnecessary data
function getDetails() public view returns (uint, uint, string memory) {
    uint id = 1;
    uint balance = 1000;
    string memory name = "User";

    // Unnecessary data like 'id' might not be needed in every context
    return (id, balance, name);
}

// Optimized function returning only necessary data
function getBalance() public view returns (uint) {
    uint balance = 1000;
    return balance;
}
```

High Fix-to-Find Ratio Vulnerabilities. In addition to the most frequently fixed vulnerabilities, we focused on vulnerabilities with a fix-to-find ratio of more than 50%, as these are consistently addressed whenever identified. Specifically, the following vulnerabilities meet this criterion:

- **Incorrect Function Signature:** Among vulnerabilities with a high fix-to-find ratio, incorrect function signatures stand out with a 66.67% fix rate. This issue occurs when a function's name or parameters do not match what is expected by other parts of the contract or by interfacing contracts, leading to operational failures or incorrect function calls. If unresolved, incorrect function signatures can cause significant operational failures, especially in contracts that interact with external systems or other contracts. The fix typically involves adjusting the function signature to ensure consistency with its intended use, as shown in Listing 11. This vulnerability is fixed consistently because it is easy to detect, often through compiler errors or during testing, and the necessary corrections are usually straightforward.

Listing 11. Incorrect function signature

```
// Incorrect function signature
function transfer(address to, uint256 amount) public {
    // Transfer logic
}

// Correct function signature to match expected interface
function transfer(address recipient, uint256 value) public {
    // Transfer logic
}
```

- **Incorrect Array Handling**: This is another vulnerability with high fix rate (50%). Incorrect array handling can lead to vulnerabilities like out-of-bounds access or unexpected behavior. This can cause runtime errors or allow attackers to exploit the contract by manipulating array indices. Listing Listing 12 shows an example of vulnerable code with potential out-of-bounds access. In this example, the lack of bounds checking on the `index` variable can lead to out-of-bounds access, potentially causing the contract to behave unexpectedly or fail. As shown in Listing 13, the fix involves adding proper bounds checks to ensure safe array access. Given the potential for severe consequences if left unfixed, this issue is addressed whenever detected, leading to its high fix-to-find ratio.

Listing 12. Vulnerable code with potential out-of-bounds access

```
// Vulnerable code with potential out-of-bounds access
function getArrayElement(uint index) public view returns (uint) {
    uint;
    return array[index]; // No bounds check
}
```

Listing 13. Fixed code with proper bounds checking

```
// Fixed code with proper bounds checking
function getArrayElement(uint index) public view returns (uint) {
    uint;
    require(index < array.length, "Index out of bounds");
    return array[index];
}
```

RQ3. What are the most common vulnerabilities fixed while upgrading smart contracts?

Answer: The results indicate that developers prioritize fixing vulnerabilities that directly impact smart contracts' core functionality, security, and economic efficiency. This prioritization includes addressing issues related to gas consumption, proper function execution, and ensuring that contracts adhere to expected data handling and access control standards. Vulnerabilities such as gas-related issues, permission controls, and visibility settings (SWC-100 and SWC-108) are frequently fixed because they can significantly affect the financial aspect of a contract, making transactions more expensive or less secure. These vulnerabilities are also relatively straightforward to fix, encouraging developers to address them promptly during upgrades. Additionally, while vulnerabilities like incorrect function signatures and array handling are less common, they are consistently resolved when identified due to their critical role in maintaining smart contracts' correct operation and interoperability.

4.5 RQ4: Analysis of Prevalent Vulnerabilities Introduced in Smart Contract Upgrades

During the upgrade process of smart contracts, developers often introduce new vulnerabilities, either due to the addition of new features or modifications to existing logic. Based on the results from RQ2, we focused on the vulnerabilities introduced in **23.35%** of smart contract versions to understand the nature of these introduced issues. The total number of introduced vulnerabilities amounted to **2,387**, encompassing **41** unique types. This section discusses the most commonly introduced vulnerabilities and those with a high introduced-to-find ratio, providing insights into why these vulnerabilities are often introduced during upgrades.

Top Introduced Vulnerabilities. Interestingly, the top vulnerabilities that emerged during the upgrade process are nearly identical to those most frequently fixed, as shown in This suggests a cyclical pattern where the issues developers strive to eliminate in upgrades are often reintroduced in subsequent versions. Figure 6 presents a bar chart illustrating these vulnerabilities. The x-axis represents the different types of vulnerabilities, while the y-axis shows the number of times each vulnerability was introduced. The most commonly introduced vulnerabilities, such as Gas-Related Issues, Permission Issues, and Visibility Problems (SWC-100 and SWC-108), often arise when new functionalities are added, or existing logic is modified without fully considering the broader implications. Figure 6.

For instance, Gas-Related Issues may be introduced when additional loops or computationally intensive operations are added without proper optimization, leading to increased gas consumption. Visibility issues (SWC-100 and SWC-108) are frequently reintroduced when new functions or state variables are created without explicitly defining their visibility, increasing the risk of unintended exposure. Permission Issues are introduced when new functions requiring access control are implemented without authorization checks, potentially leaving the contract vulnerable.

High Introduced-to-Find Ratio Vulnerabilities. In addition to the most commonly introduced vulnerabilities, certain vulnerabilities exhibit a high introduced-to-find ratio, meaning they are more likely to emerge in later versions of the contract rather than in the initial deployment. We focused on vulnerabilities with a ratio of more than 50%, which are:

- **SWC-103: DoS with Failed Call** is a vulnerability with a high introduced-to-find ratio of 62.5%. This issue occurs when a contract does not properly handle failed external calls, leading to potential denial of service (DoS). This vulnerability is often introduced in later contract versions when developers expand the contract's reliance on external interactions or add new functionalities without implementing necessary error-handling mechanisms. The assumption that external calls will always succeed without fallback mechanisms can lead to this vulnerability, making it more common in upgraded contract versions.
- **SWC-105: Unprotected Ether Withdrawal** also has a high introduced-to-find ratio of 50.1%. This vulnerability arises when developers fail to properly secure Ether withdrawal functions, which could allow unauthorized entities to withdraw funds. This issue is frequently introduced in later versions of the contract when new withdrawal mechanisms are added, or existing ones are modified, and the necessary access control checks are either overlooked or inadequately implemented.

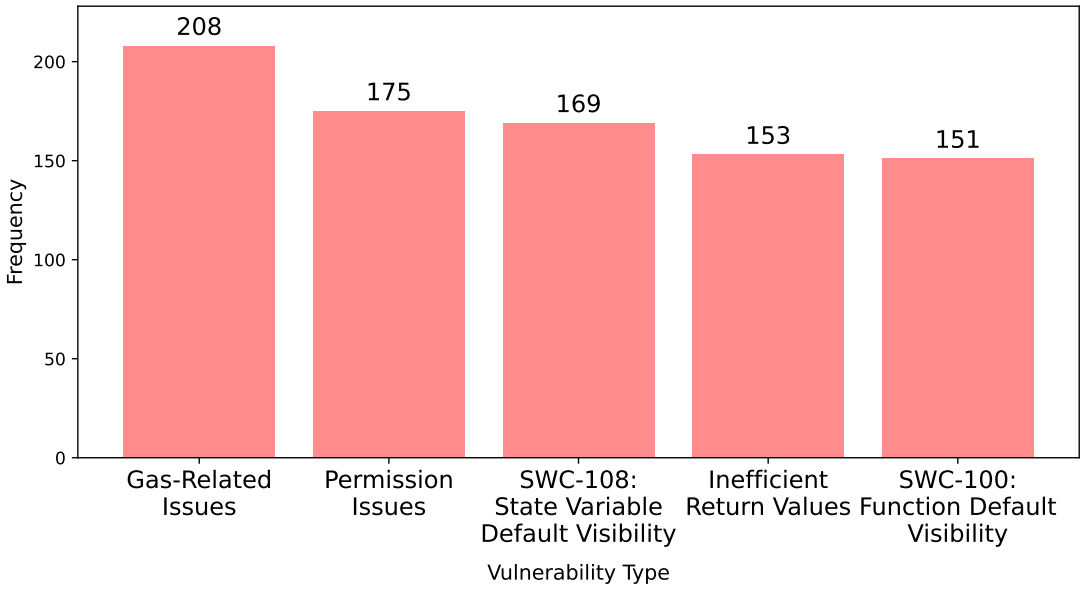

Fig. 6. Top common vulnerabilities introduced in smart contract upgrades(RQ4)

RQ4. What are the most common vulnerabilities introduced while upgrading smart contracts?

Answer: The analysis revealed a total of 41 unique vulnerabilities introduced during smart contract upgrades. Intriguingly, the top vulnerabilities introduced during upgrades closely mirror those most frequently fixed. This overlap underscores the persistent nature of these issues and the cyclical challenge they present in smart contract development. Despite developers' efforts to address vulnerabilities such as Gas-Related Issues, Permission Controls, and Visibility Settings (SWC-100 and SWC-108), these issues tend to recur in later versions of contracts. This cycle highlights the inherent complexity of maintaining secure and efficient smart contracts, where introducing new features or modifications to existing logic often reintroduces previously resolved vulnerabilities. Additionally, vulnerabilities with high introduced-to-find ratios, such as SWC-103 (DoS with Failed Call) and SWC-105 (Unprotected Ether Withdrawal), underscore the risks of expanding contract functionality without adequately addressing potential security implications. This indicates specific areas where the introduction of issues is more likely due to the nature of the contract's evolving functionality.

4.6 RQ5: Persistence of Vulnerabilities in Smart Contract Upgrades

To answer the research question, we identified a total of 670 persistent vulnerabilities across 33 unique types in smart contract upgrades. These vulnerabilities were analyzed to determine the challenges and reasons behind their persistence despite upgrade efforts. We analyzed two specific scenarios from RQ2 to understand the persistence and challenges in remediating vulnerabilities in smart contract versions, which are: (1) vulnerabilities that persisted without any being fixed pr introduced (Negative Impact Cases), and (2) vulnerabilities that remained despite some being fixed.

Case 1 - Negative Impact Cases: These involve contract versions where no new vulnerabilities were introduced, none were fixed, yet unresolved vulnerabilities persist. The top dominant vulnerabilities in this category are:

- **Gas-Related Issues:** These persist in 16.27% of cases, primarily due to the complexity of optimizing gas consumption in specific contract scenarios. While some gas inefficiencies, such as those found in simple loops, are relatively easy to detect and remediate, more complex issues present significant challenges. For instance, consider a contract that involves dynamic storage operations, where the cost of reading and writing to storage can vary significantly depending on the state of the contract. Unlike a straightforward loop, optimizing gas consumption in this context might require an intricate redesign of the storage management logic. Developers may hesitate to make such changes due to the risk of introducing new vulnerabilities or breaking existing functionality, leading to the persistence of these gas-related issues across versions.
- **Permission Issues:** These persist in 10.15% of cases and are often related to the difficulty of implementing comprehensive access control across all functions within a contract. Simple access control issues, such as adding a modifier to restrict function access, are easy to resolve. However, ensuring that all functions have appropriate access controls can be challenging in more complex contracts, especially those that involve multiple inheritance layers or external dependencies. For example, a function might inherit insufficient access control from a parent contract. Addressing such issues may require a thorough review of the entire contract hierarchy, a process that is often overlooked or inadequately addressed, leading to persistent permission vulnerabilities.
- **SWC-102: Outdated Compiler Version:** This issue occurs in 9.85% of cases. Although upgrading the compiler version is technically straightforward, developers often resist doing so due to concerns about breaking existing functionality or introducing new, unexpected behaviors. For example, a contract that relies on specific quirks or behaviors of an older compiler version might not function correctly if recompiled with a newer version. This cautious approach to compiler upgrades can result in the continued use of outdated versions, leaving associated vulnerabilities unaddressed.
- **Incorrect Coding Practices:** These persist in 8.81% of cases because they are often deeply embedded in the contract's codebase and may be overlooked during development and code reviews. Incorrect coding practices refer to poor coding standards that can introduce vulnerabilities or result in inefficient contract execution. Examples include the misuse of functions, improper data handling, and failure to adhere to best practices. For instance, consider a contract where functions are poorly named or inconsistently structured, leading to confusion and potential errors. Refactoring such a contract to follow best practices would require a significant time investment and might disrupt the contract's operation. As a result, developers may deprioritize these fixes, allowing incorrect coding practices to persist across versions.
- **Inefficient Return Values:** These persist in 8.21% of cases due to the challenges involved in optimizing function outputs, particularly when dealing with complex data structures or large arrays. These issues can be fixed in simpler cases by adjusting the function to return only the necessary data. However, in more complex scenarios, optimizing return values might require significant refactoring of multiple interdependent functions. For example, reducing a function's size without disrupting the contract's logic or functionality can be difficult if a function returns a large, nested data structure. Developers may avoid making these optimizations, fearing unintended side effects or increased code complexity, resulting in the persistence of inefficient return values.

Case 2 - Fixed Vulnerabilities with Remaining Issues: These involve cases where certain vulnerabilities were addressed, but others persisted. The top vulnerabilities in this scenario are:

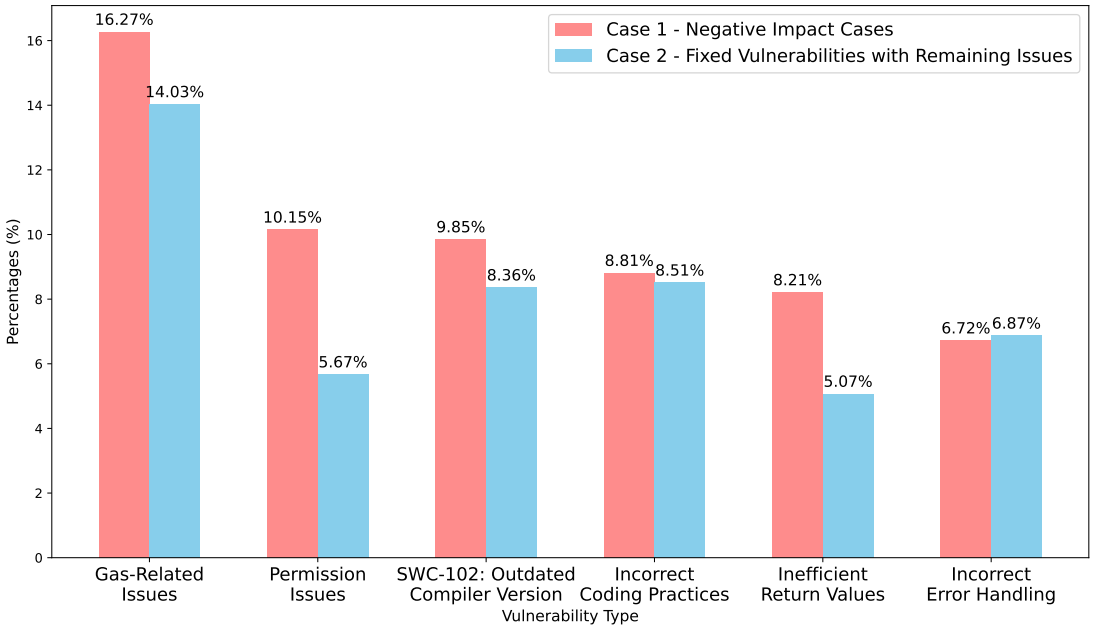

Fig. 7. Persistence of vulnerabilities in smart contract upgrades (RQ5)

- **Gas-Related Issues:** These persist in 14.03% of cases. As previously discussed, these issues persist due to their complexity.
- **Incorrect Coding Practices:** These remain an issue in 8.51% of cases due to deep integration into the contract's structure.
- **SWC-102: Outdated Compiler Version:** This issue remains in 8.36% of cases. As previously noted, developers may avoid upgrading the compiler to prevent potential disruptions.
- **Incorrect Error Handling:** This persists in 6.87% of cases, often because it is overlooked in favor of addressing more critical vulnerabilities. Poor error handling can lead to contracts failing silently or behaving unexpectedly, creating security risks. For instance, if a function fails to check for error conditions properly, it might continue executing with incorrect assumptions, leading to unexpected results or security vulnerabilities. Ensuring robust error handling across all scenarios can be challenging in complex contracts with many interdependent functions. Developers might address obvious error-handling issues but miss those that occur under specific, less common conditions, leading to the persistence of these vulnerabilities.
- **Permission Issues:** These persist in 5.67% of cases. Simple permission issues can be resolved easily, but more complex access control scenarios often go unaddressed due to the potential risks involved in modifying contract logic.

Figure 7 presents a comparative bar chart that visualizes the prevalence of key vulnerabilities in smart contracts across two scenarios: Case 1 - Negative Impact Cases and Case 2 - Fixed Vulnerabilities with Remaining Issues. The x-axis lists the types of vulnerabilities, while the y-axis represents the percentage of contracts affected by each vulnerability in the

respective cases. Figure 7 shows that while some vulnerabilities, such as permission issues, tend to be mitigated when other vulnerabilities are addressed, others, such as gas-related inefficiencies and outdated compiler versions, often persist across contract versions. The reductions observed from Case 1 to Case 2 suggest that remediation efforts can lead to partial improvements, yet the consistent presence of certain vulnerabilities highlights the ongoing challenges in achieving comprehensive security in smart contract upgrades.

RQ5. Do some vulnerabilities persist across versions despite the upgrade efforts?

Answer: Yes, some vulnerabilities do persist across versions despite upgrade efforts. This persistence can be attributed to several factors. First, the inherent complexity of certain vulnerabilities, such as those involving intricate gas optimizations or deeply embedded coding practices, often requires extensive refactoring or even architectural changes. These modifications are labor-intensive and carry the risk of introducing new errors or altering the intended behavior of the contract. As a result, developers might hesitate to make such changes, potentially leading to the persistence of these issues. However, it is also possible that some vulnerabilities persist due to being overlooked or deprioritized, especially if they are less visible or perceived as less critical. Without direct insight into the development processes, it's difficult to determine whether these issues were intentionally left unresolved or simply missed during upgrades. Therefore, the persistence of vulnerabilities could be due to a combination of technical challenges, oversight, or selective prioritization, which together contribute to ongoing risks in smart contract security.

5 Discussion

In this section, we discuss the persistence of vulnerabilities in smart contracts, focusing on how issues like Gas-Related Problems and Permission Issues reoccur across different contract versions. We examine the lifecycle of these vulnerabilities to understand the difficulties in eliminating them. Additionally, we address the difference between vulnerabilities that are present in the code and those that are actually exploitable.

5.1 Lifecycle of Persistent Vulnerabilities in Smart Contracts

Following the insights gained from RQ3 and RQ4, where Gas-Related Issues and Permission Issues emerged as the most commonly fixed and introduced vulnerabilities, we conducted a deeper analysis to understand the lifecycle of these vulnerabilities across multiple versions within the same proxy contract. The goal was to explore whether these vulnerabilities, once introduced and fixed, tend to reappear in later versions, and to identify any patterns in their reintroduction and subsequent fixes. Understanding these patterns is crucial for improving smart contract security practices, as it highlights areas where current approaches may be insufficient and where more robust solutions may be required. To achieve this, we analyzed the following metrics:

- **Introduced in One Version and Fixed in a Later Version:** This metric tracks instances where a vulnerability was introduced in one version of a proxy contract and then fixed in a subsequent version, helping us understand how often vulnerabilities persist across multiple versions before being addressed.
- **Fixed in One Version and Reintroduced in a Later Version:** This metric identifies cases where a vulnerability was fixed in one version but reappeared in a later version, critical for understanding the resilience of fixes and whether vulnerabilities tend to recur.
- **Reintroduced After Fix:** This counts every instance of a vulnerability being reintroduced after being fixed, providing a cumulative view of how persistent a vulnerability is across the contract's lifecycle.

- **Fixed After Reintroduction:** This measures how often a vulnerability that was reintroduced after being fixed was subsequently addressed again in later versions, highlighting the effectiveness of repeated fixes.

Gas-Related Issues show a complex lifecycle across different versions of the same proxy contract. With 31 occurrences where this vulnerability was introduced in one version and fixed in a subsequent version, and 17 cases where it was fixed and then reintroduced, it is evident that Gas-Related Issues are persistent challenges. The cumulative count of 52 reintroductions after a fix indicates that even after being addressed, these issues often resurface as the contract evolves. This persistence likely stems from the inherent difficulty in optimizing gas usage, which can be affected by even minor changes in the contract. As new features are added or existing ones are modified, the gas efficiency of the contract can degrade, leading to the reintroduction of this vulnerability. Additionally, the 10 instances where the vulnerability was reintroduced and then fixed again suggest that developers must repeatedly address these issues as the contract matures.

Permission Issues follow a similar pattern, with 30 occurrences of being introduced and later fixed, and 14 instances of being fixed and then reintroduced. The 65 occurrences of reintroduction after being fixed highlight the ongoing challenge of maintaining robust access controls. Unlike Gas-Related Issues, there were no recorded instances where Permission Issues were fixed again after reintroduction, which may indicate that these vulnerabilities, once reintroduced, are not always immediately recognized or addressed. This could be due to the complexity of implementing comprehensive permission systems, where developers may inadvertently introduce gaps in access control as they update and modify the contract.

In contrast, **SWC-100: Function Default Visibility** and **SWC-108: State Variable Default Visibility** did not show any occurrences of being introduced and then fixed across different versions, nor were there any cases of reintroduction after a fix. This difference can be attributed to the nature of these vulnerabilities. Visibility issues are straightforward to identify and correct—once a function or state variable's visibility is set, it typically doesn't require further changes unless the contract undergoes significant restructuring. The simplicity of these fixes, combined with the fact that visibility settings are a fundamental part of Solidity's best practices, means that developers are likely addressing these issues immediately and effectively when they arise. Moreover, the absence of these vulnerabilities in the lifecycle analysis suggests that their occurrence may be more closely tied to the coding standards and expertise of the developers involved. Developers who are well-versed in Solidity's best practices are likely to avoid introducing these issues in the first place, or they correct them as soon as they are detected.

Inefficient Return Values also did not show any occurrences of being introduced and fixed across different versions. This may reflect the nature of this vulnerability, which typically arises when functions return more data than necessary, leading to higher gas costs and potential exposure of unnecessary information. Like visibility issues, once identified, the fix for Inefficient Return Values is usually straightforward—developers simply modify the function to return only the necessary data. However, the absence of this vulnerability in the lifecycle analysis may also indicate that different developers have varying focuses and priorities. For some, optimizing return values may not be a primary concern, especially if the contract is not highly gas-sensitive. Alternatively, it could reflect a more deliberate approach where developers optimize return values only when the inefficiency becomes a noticeable issue, rather than as a standard practice during initial development.

The contrast between these vulnerabilities and the recurring patterns observed in Gas-Related Issues and Permission Issues highlights the different levels of complexity and developer focus required to manage them. Gas-Related Issues and Permission Issues are inherently more dynamic and closely tied to the functional and security-critical aspects of

smart contracts, making them more prone to recurrence as the contract evolves. On the other hand, SWC-100, SWC-108, and Inefficient Return Values are more static, likely influenced by the developer's adherence to coding standards and best practices. This difference underscores the importance of ongoing vigilance for certain types of vulnerabilities, while also suggesting that the consistent application of best practices can effectively mitigate others.

5.2 Vulnerable Code vs. Exploitable

The study focused on the detected vulnerabilities; however, this does not mean that these vulnerabilities are exploited. The distinction between exploitable and vulnerable code is crucial in understanding the implications of detected vulnerabilities in smart contracts. While all exploitable vulnerabilities are inherently vulnerable, not all vulnerabilities can be exploited. This distinction is critical because the mere presence of a vulnerability does not necessarily imply an immediate security threat. Zhang et al. [46] and Perez et al. [47] emphasize that although many smart contracts contain vulnerabilities, only a subset of these are exploited in practice, and this exploitation is often dependent on specific conditions such as the Ether balance of the contract.

To investigate whether these vulnerabilities have been exploited, a comprehensive analysis of the transactional activity associated with these contracts is essential. Such an analysis can reveal patterns of exploitation or demonstrate that some vulnerabilities, while present, have not been actively exploited. However, this type of analysis is non-trivial and was not conducted in this work due to its complexity and the extensive data required.

Our dataset reveals that the most frequently detected vulnerabilities, such as Gas-Related Issues and Permission Issues, are not typically classified as directly exploitable. Gas-related issues, for example, lead to inefficiencies like high transaction costs or failed transactions due to excessive gas consumption. Similarly, Permission Issues, though not directly exploitable in many cases, can compromise a contract's functionality if access controls are not properly implemented, potentially allowing unauthorized access to contract functions. While these vulnerabilities may not lead to direct exploitation, they pose substantial risks to the operational efficiency and reliability of smart contracts, underscoring the importance of addressing them even if they do not present an immediate security threat.

In contrast, our analysis identified several vulnerabilities that are commonly recognized as exploitable. These include:

- **SWC-107: Reentrancy** – A vulnerability allowing attackers to repeatedly call a function within a contract before the initial execution is complete, potentially draining funds.
- **SWC-106: Unprotected SELFDESTRUCT Instruction** – If a contract allows self-destruction without proper authorization checks, it can be exploited to delete the contract or redirect funds.
- **SWC-105: Unprotected Ether Withdrawal** – This issue arises when a contract does not properly secure its Ether withdrawal functions, potentially allowing unauthorized users to withdraw funds.
- **SWC-104: Unchecked Call Return Value** – Failing to check the return value of a low-level call can lead to unintended consequences, particularly if the called contract fails.
- **SWC-112: Delegatecall to Untrusted Callee** – A vulnerability that allows arbitrary code execution in the context of the calling contract, posing significant security risks if the callee contract is untrusted.
- **SWC-124: Write to Arbitrary Storage Location** – This vulnerability allows attackers to modify critical data in the contract's storage, potentially leading to severe consequences.
- **SWC-113: DoS with Failed Call** – An attacker could prevent the contract from functioning correctly by forcing specific calls to fail, causing a denial of service.

Table 4. Percentage of Commonly Exploitable Vulnerabilities

Vulnerability	% of Total Detected	% of Total Fixed (of Detected)	% Unfixed in Last Version (of Detected)
SWC-107: Reentrancy	1.95%	65.38%	34.62%
SWC-106: Unprotected SELFDESTRUCT	0.06%	50.00%	50.00%
SWC-105: Unprotected Ether Withdrawal	0.15%	80.00%	20.00%
SWC-104: Unchecked Call Return Value	3.36%	73.66%	26.34%
SWC-112: Delegatecall to Untrusted Callee	3.13%	77.03%	22.97%
SWC-124: Write to Arbitrary Storage	0.13%	66.67%	33.33%
SWC-113: DoS with Failed Call	0.12%	75.00%	25.00%

It is important to note that not every instance of these vulnerabilities will necessarily lead to exploitation. The exploitability of a vulnerability depends on various factors, including the specific conditions required for exploitation and whether additional security measures are in place. For example, while Reentrancy (SWC-107) is a well-known vulnerability, it becomes exploitable primarily when a contract allows external calls to untrusted contracts within a function that modifies the contract's state. If a contract does not perform external calls during critical state changes or carefully sequences these interactions, the presence of a Reentrancy vulnerability may not lead to an exploit. Similarly, Unchecked Call Return Value (SWC-104) may only lead to an exploit if the contract assumes the success of an external call without verifying it, particularly when that call is crucial to the contract's functionality.

Table **??** presents the percentages of these commonly exploitable vulnerabilities relative to the total number of vulnerabilities detected for each proxy contract. The percentages of fixed and unfixed vulnerabilities in the table are calculated as a proportion of the detected vulnerabilities for each category, rather than the total vulnerabilities in the dataset.

Table 4 illustrates several critical points. The total percentages of commonly exploitable vulnerabilities relative to all detected vulnerabilities are relatively low. This suggests that while these vulnerabilities are serious, they represent a small fraction of the overall vulnerabilities detected in smart contracts.

The relatively low overall percentage of commonly exploitable vulnerabilities in the dataset could be attributed to several key factors. One significant factor is the developers' focus on these specific vulnerabilities due to their high-profile nature and the availability of effective detection tools. Tools designed to identify high-impact issues, such as Reentrancy (SWC-107), are widely used. Given that Reentrancy is one of the most well-known vulnerabilities in smart contracts, developers are likely to prioritize mitigating it from the outset, reducing its presence from the beginning of the development process. Additionally, the well-documented nature of these vulnerabilities and the availability of clear fixing patterns, such as the 'Checks-Effects-Interactions' pattern for Reentrancy, contribute to their early detection and resolution. This proactive approach ensures that such vulnerabilities are less likely to appear or persist in the final versions of smart contracts.

Additionally, while the dataset includes many common vulnerabilities, it does not cover all exploitable types. For example, **SWC-114: Transaction Order Dependence** and **SWC-120: Weak Sources of Randomness from Chain Attributes** are known exploitable vulnerabilities that were not detected in this dataset. These vulnerabilities are particularly challenging to detect because they depend on specific execution contexts or rely on external factors like block attributes. Zhang et al. [46] highlight that many current detection tools may not effectively identify these

vulnerabilities because they require dynamic analysis or specific environmental conditions that are not easily simulated in static analysis.

However, it is important to emphasize that although the current data suggests a lack of exploitation for many detected vulnerabilities, this does not mitigate the potential risks they pose. Without proper mitigation measures, vulnerabilities remain susceptible to exploitation in the future. Therefore, continued caution and proactive measures are necessary to address vulnerabilities in smart contracts effectively.

6 Future Research Directions

In light of the findings from this study, several avenues for future research are apparent, focusing on improving the management of vulnerabilities across different versions of smart contracts and exploring the utility of the dataset generated in this research.

One key area for future exploration is how different upgrade strategies impact the introduction and resolution of vulnerabilities. For instance, understanding the effects of modularization or proxy patterns on vulnerability management could provide valuable insights. Research could investigate whether certain upgrade practices inadvertently lead to the reintroduction of vulnerabilities and how these risks might be mitigated through improved design patterns or coding practices. Another important aspect to consider is the role of developer experience and expertise in the occurrence and management of vulnerabilities. Empirical studies comparing smart contracts developed by teams with varying levels of experience could determine whether more experienced developers are less likely to introduce certain vulnerabilities or if they are more effective in preventing the reintroduction of previously resolved issues. This could inform targeted training and educational programs to enhance smart contract developers' security practices. The relationship between detected vulnerabilities and their exploitability in real-world scenarios warrants further investigation. While static analysis tools can identify potential vulnerabilities, understanding which of these are likely to be exploited remains a challenge. Future research could focus on developing predictive models that assess the likelihood of a vulnerability being exploited based on factors such as contract complexity, coding patterns, and historical upgrade data. These models would be instrumental in prioritizing security efforts, enabling developers to focus on the most critical issues that pose genuine risks to contract integrity.

The dataset generated from this study holds significant potential for machine learning applications. By training models on this historical data, researchers can create predictive systems that identify likely points of failure in new smart contracts before they are deployed. Such systems could enhance the detection of vulnerabilities and streamline the process of managing vulnerabilities in a scalable manner, making the development of secure smart contracts more efficient. Furthermore, the dataset could serve as a benchmark for evaluating new tools and methods designed to detect and mitigate vulnerabilities. Researchers can use this dataset to assess the effectiveness of these tools in real-world scenarios, ensuring they are robust enough to handle the complexities of smart contract development. This could lead to the development of more sophisticated tools that better address the specific challenges identified in this study. Lastly, the dataset can also be used to create practical resources for developers. By analyzing patterns in the data, it might be possible to develop guidelines or checklists that help developers avoid common pitfalls during upgrades. Additionally, integrating this dataset into development environments could provide real-time feedback to developers, alerting them to potential vulnerabilities as they code, thereby reducing the likelihood of introducing new issues.

7 Related Work

This study presents an empirical analysis of the impact of upgrading smart contracts on security, a topic that has not been extensively explored in the existing literature. We categorize related work into three main areas: vulnerability datasets for smart contracts, smart contract upgrades, and the impact of traditional software upgrades on security. We highlight how our study distinguishes itself from these existing works by offering a dynamic perspective on how security evolves over time with contract upgrades.

7.1 Smart Contract Vulnerability Datasets

Research on smart contract vulnerabilities has focused extensively on developing datasets that identify, categorize, and analyze various security issues within Ethereum contracts. These datasets provide a foundation for understanding the prevalence and characteristics of different vulnerabilities.

Durieux et al. [29] conducted an empirical analysis of automated tools applied to a large dataset of Ethereum smart contracts, evaluating their effectiveness in detecting vulnerabilities and highlighting the strengths and weaknesses of various approaches. Yashavant et al. [44] created the `Scrawld` dataset, which includes labeled smart contracts from Etherscan, identifying vulnerabilities such as reentrancy, timestamp dependence, and gas-related issues to support the development of automated detection techniques. Ren et al. [48] compiled a dataset that includes real-world and artificially constructed vulnerable contracts to evaluate smart contract testing methods, enabling comparative analysis of different testing approaches. Zhang et al. [43] developed a framework and dataset targeting bugs in Ethereum smart contracts, facilitating both static and dynamic analysis of vulnerabilities. Soud et al. [45] introduced `Automesc`, a framework that mines and classifies vulnerabilities in Ethereum smart contracts and tracks their resolution over time using GitHub commit data, providing insights into how vulnerabilities evolve and are addressed. Zheng et al. [49] developed `DappScan`, a tool designed to build large-scale datasets of smart contract weaknesses in decentralized applications (DApps), focusing on security issues unique to these applications. Cai et al. [50] proposed an approach for fine-grained vulnerability detection through heterogeneous code feature learning and automated dataset construction, contributing to developing advanced detection algorithms.

These datasets offer valuable insights into the static characteristics of smart contract vulnerabilities; they do not specifically address how these vulnerabilities evolve across different versions due to upgrades. Our work constructs a dataset that tracks smart contract versions over time, detailing the vulnerabilities present in each version. This enables an empirical analysis of how contract upgrades affect the introduction, resolution, or persistence of vulnerabilities, providing a dynamic perspective that complements existing datasets.

7.2 Smart Contract Upgrades

Existing research on smart contract upgradeability has focused on developing new frameworks to support contract upgrades, evaluating current upgrade practices, detecting upgrade approaches, and examining the security issues associated with these approaches.

Several studies have introduced new models to enhance the upgradeability of smart contracts. Bui et al. [51] proposed a data-proxy model that allows changes in contract logic while preserving data integrity, thereby maintaining security properties even when the contract is modified. Girish and Kaganurmath [52] introduced the "Comprehensive-Data-Proxy pattern," which uses data segregation over the proxy pattern to defend against reentrancy attacks and mitigate scalability issues inherent to proxy patterns. Their experiments demonstrated that this framework effectively addresses these

concerns with minimal performance impact. Ferreira [53] developed a framework for securely upgrading Ethereum smart contracts, emphasizing careful management of contract state and storage to prevent vulnerabilities during upgrades. Other studies have analyzed existing upgrade practices to understand their effectiveness and challenges. Marin [54] explored how upgradeable contracts can be applied in decentralized applications (DApps), identifying patterns that help address immutability while maintaining contract integrity. Salehi [55] examined different upgrade mechanisms, such as proxy contracts, in Ethereum-based applications, particularly in decentralized finance (DeFi), highlighting the balance between flexibility and security. We conducted an empirical study on the Ethereum blockchain [56] to investigate the use of proxy patterns in real-world upgrade practices, identifying common challenges and risks associated with these mechanisms.

Research has also focused on detecting upgrade mechanisms used in smart contracts. Bodell et al. [57] developed a comprehensive taxonomy for characterizing the unique behaviors of upgradeable smart contracts (USCs) and created USCHUNT, an automated analysis framework to support their study. Ebrahimi et al. [58] introduced the UPC Sentinel method to accurately detect proxy contracts on Ethereum, which is important for maintaining security during the upgrade process.

Recently, there has been increased attention on the security issues related to existing upgrade mechanisms. Huang et al. [59] analyzed the risks of using proxy contracts for upgrades, particularly the difficulty of maintaining immutability while allowing modifications. Liu et al. [60] and Li et al. [61] examined the security impact of different upgrade patterns on Ethereum, identifying potential risks and recommending best practices to enhance contract security.

Existing studies provide a broad understanding of smart contract upgradeability and its associated challenges, primarily focusing on proposing new frameworks, analyzing current practices, detecting upgrade mechanisms, or assessing general security risks. Our study differs by empirically examining how smart contract upgrades affect security over time, specifically analyzing how upgrades impact the vulnerability profile of smart contracts, whether they resolve existing vulnerabilities, introduce new ones, or leave some issues unresolved.

7.3 Traditional Software Upgrade Impact on Security

Research on the security impact of software upgrades in traditional software systems provides valuable insights relevant to understanding similar challenges in smart contracts despite the latter's unique vulnerabilities and considerations due to their decentralized and immutable nature.

Studies have shown that software patches and version upgrades, while necessary to enhance security, can introduce new issues due to changes in software behavior, dependencies, or interactions with existing components. Oberheide et al. [62] and Treetippayaruk and Senivongse [63] found that software upgrades while fixing known vulnerabilities, could inadvertently create new vulnerabilities through unforeseen interactions or dependencies. This is comparable to our findings, where smart contract upgrades aimed at improving security often introduce new vulnerabilities, such as gas-related issues and improper function visibility, which are specific to smart contracts.

Dissanayake et al. [64] highlighted the complexities of managing software security patches and emphasized the need for rigorous testing and validation to prevent new vulnerabilities. This aligns with our observation that smart contract upgrades frequently fail to address all existing vulnerabilities and may introduce new ones if not thoroughly tested. The persistence of certain vulnerabilities in our dataset, such as gas-related inefficiencies and incorrect coding practices, demonstrates the need for comprehensive testing and validation during the upgrade process.

Wash et al. [65] and Vaniea et al. [66] discussed the unintended security consequences that can arise from automated software updates. Similarly, our study shows that smart contract upgrades can have unexpected security impacts,

particularly when changes are made without adequate consideration of the broader security implications. This finding reinforces the need for careful management of the upgrade process to prevent new vulnerabilities from emerging.

Dumitraş et al. [67] examined how upgrade decisions impact security across domains, noting the trade-offs between security, functionality, and operational considerations. Our study confirms that similar trade-offs exist in smart contracts, where upgrades must balance the need to enhance functionality with the risk of introducing new security vulnerabilities. This is evident from our findings, where fixing certain vulnerabilities often coincides with the emergence of new ones, reflecting the complexity of maintaining a secure contract over multiple versions.

While these studies from traditional software domains provide important insights into the general risks associated with software upgrades, they do not fully address the specific challenges of smart contracts. Smart contracts face unique issues such as gas optimization, immutable code constraints, and decentralized execution environments that require specialized approaches. Our study builds on the understanding that software upgrades impact security by empirically analyzing how these effects manifest uniquely in smart contracts, particularly concerning the introduction, remediation, or persistence of vulnerabilities over multiple versions.

8 Threats to Validity

Several potential threats to the validity of our study were identified and addressed to ensure the robustness of our findings. These threats are categorized into internal and external validity concerns.

8.1 Internal Validity

One of the primary internal validity threats arises from the reliance on automated security analysis tools, which are known to generate false positives or overlook certain vulnerabilities. Although we applied filtering strategies and a majority rule to mitigate these risks, the possibility remains that some vulnerabilities may not have been accurately detected or classified. The tools selected for this study were chosen based on their extensive validation in both academic and industry contexts. However, the inherent limitations of these tools may still impact the accuracy of our vulnerability detection and classification process.

Another significant internal validity threat involves the process of identifying and categorizing smart contracts and their versions. While the methodology was designed to systematically identify upgradeable proxy contracts and extract their versions, there is a potential risk of misclassification or inaccuracies in identifying these contracts. Although manual labeling was avoided to reduce human error, the automated processes used might still introduce inaccuracies. To address this, extensive testing and validation of each methodological step were conducted to ensure accuracy and reliability, minimizing the risk of errors in contract version identification and categorization.

A further internal validity concern relates to using a 0.5 similarity threshold for fuzzy and structural AST comparisons. This threshold was chosen based on experimentation to balance flexibility and precision. A higher threshold could miss meaningful similarities from minor changes, while a lower threshold might allow too many false positives. In some cases, distinct functions with similar structures (e.g., similar control flows or event emissions) may exceed the threshold and be misclassified as similar. While manual review of such cases may help, it is not always feasible or efficient in large-scale analyses. Therefore, we selected the 0.5 threshold to minimize false negatives while acknowledging the possibility of some false positives. This trade-off represents a potential threat to the accuracy of our findings. Further refinements to the automated process may improve accuracy in future iterations.

Another potential threat is the occasional absence of certain smart contract versions due to missing source code on platforms like Etherscan. In such cases, we compared the vulnerabilities between the current version and the next

available version. Although the frequency of missing versions was low, this approach could lead to gaps in the analysis, potentially resulting in an incomplete understanding of vulnerability evolution. However, given the limited occurrence of such cases, we believe the overall impact on the study's conclusions is minimal.

8.2 External Validity

The primary external validity threat concerns the generalizability of our findings beyond the specific context of Ethereum smart contracts. Our study exclusively focuses on Ethereum, leveraging its unique characteristics and ecosystem, which may limit the applicability of our results to other blockchain platforms or smart contract types. While our methodology is robust and provides significant insights into the security implications of smart contract upgrades within the Ethereum ecosystem, it may not fully apply to smart contracts on different platforms due to differences in underlying technologies, languages, and development practices.

Additionally, the focus on proxy-based upgradeable contracts further narrows the generalizability of our findings. The specific challenges and vulnerabilities associated with this upgrade pattern may not be relevant to other smart contract designs that do not use proxies or have different upgrade mechanisms. As such, the conclusions drawn from this study should be interpreted within the specific context of Ethereum-based, proxy-upgradeable contracts.

Despite these limitations, the careful application of our methodology, supported by extensive testing on a diverse sample of Ethereum contracts, provides confidence in the internal validity of our findings. However, caution should be exercised when attempting to generalize these results to other contexts or blockchain environments.

9 Conclusion

This study addressed five key research questions to understand the impact of smart contract upgrades on security. Through an empirical analysis of multiple smart contract versions, we found a moderate correlation between the number of versions and the resolution of vulnerabilities, suggesting that while frequent upgrades can help fix certain security issues, they often introduce new vulnerabilities. This dual impact indicates that while upgrades are necessary for maintaining contract functionality and security, they also present a risk of creating new weaknesses. Our findings reveal that some vulnerabilities, such as gas-related issues, function visibility problems, and permission control weaknesses, are frequently fixed and reintroduced during upgrades. This cyclical pattern suggests a need for more careful upgrade strategies to prevent known vulnerabilities' recurrence. Furthermore, persistent vulnerabilities, such as gas inefficiencies and outdated compiler versions, demonstrate that current practices often struggle to fully address these issues, underscoring the need for more robust approaches to smart contract maintenance. The study also highlights that while upgrades are intended to improve security, they require more rigorous testing, validation, and strategic planning to avoid new risks. The persistence of certain vulnerabilities across multiple versions indicates that more comprehensive solutions are needed to mitigate these security challenges effectively. By addressing these five research questions, this study provides valuable insights into the complex relationship between smart contract upgrades and security. The dataset generated from this research offers a valuable resource for further exploration of vulnerability patterns, aiding in developing more secure and resilient smart contract frameworks.

References

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

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

[3] S. Sayeed, H. Marco-Gisbert, and T. Caira, "Smart contract: Attacks and protections," *IEEE Access*, vol. 8, pp. 24416–24427, 2020.

[4] B. K. Mohanta, S. S. Panda, and D. Jena, "An overview of smart contract and use cases in blockchain technology," in *2018 9th international conference on computing, communication and networking technologies (ICCCNT)*, pp. 1–4, IEEE, 2018.
[5] D. He, Z. Deng, Y. Zhang, S. Chan, Y. Cheng, and N. Guizani, "Smart contract vulnerability analysis and security audit," *IEEE Network*, vol. 34, no. 5, pp. 276–282, 2020.
[6] P. Qian, Z. Liu, Q. He, B. Huang, D. Tian, and X. Wang, "Smart contract vulnerability detection technique: A survey," *arXiv preprint arXiv:2209.05872*, 2022.
[7] N. F. Samreen and M. H. Alalfi, "A survey of security vulnerabilities in ethereum smart contracts," *arXiv preprint arXiv:2105.06974*, 2021.
[8] M. Rodler, W. Li, G. O. Karame, and L. Davi, "{EVMPatch}: Timely and automated patching of ethereum smart contracts," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1289–1306, 2021.
[9] P. Antonino, J. Ferreira, A. Sampaio, and A. Roscoe, "Specification is law: Safe creation and upgrade of ethereum smart contracts," *arXiv preprint arXiv:2205.07529*, 2022.
[10] M. Salehi, J. Clark, and M. Mannan, "Not so immutable: Upgradeability of smart contracts on ethereum," *arXiv preprint arXiv:2206.00716*, 2022.
[11] J. Chen, X. Xia, D. Lo, J. Grundy, and X. Yang, "Maintenance-related concerns for post-deployed ethereum smart contract development: issues, techniques, and future challenges," *Empirical Software Engineering*, vol. 26, no. 6, p. 117, 2021.
[12] S. Wang, Y. Yuan, X. Wang, J. Li, R. Qin, and F.-Y. Wang, "An overview of smart contract: architecture, applications, and future trends," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, pp. 108–113, IEEE, 2018.
[13] E. Androulaki *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the thirteenth EuroSys conference*, pp. 1–15, 2018.
[14] R. G. Brown, J. Carlyle, I. Grigg, and M. Hearn, "Corda: an introduction," *R3 CEV, August*, vol. 1, no. 15, p. 14, 2016.
[15] V. Buterin *et al.*, "Ethereum white paper," *GitHub repository*, vol. 1, pp. 22–23, 2013.
[16] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart contract templates: foundations, design landscape and research directions," *arXiv preprint arXiv:1608.00771*, pp. 1–1, 2016.
[17] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pp. 8–15, IEEE, 2019.
[18] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, pp. 9–16, 2018.
[19] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, pp. 653–663, ACM, 2018.
[20] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 254–269, ACM, 2016.
[21] H. Wang, H. Liu, and Y. Wang, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 737–749, ACM, 2020.
[22] T. D. Nguyen, T. H. D. Hoang, and J. Sun, "The tale of confuzzius: A data dependency-aware hybrid fuzzer for smart contracts," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 349–360, ACM, 2020.
[23] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pp. 67–82, 2018.
[24] "Mythril." https://github.com/ConsenSys/mythril.
[25] W. E. Bodell III, S. Meisami, and Y. Duan, "Proxy hunting: understanding and characterizing proxy-based upgradeable smart contracts in blockchains," in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 1829–1846, 2023.
[26] I. Qasse, M. Hamdaqa, and B. Þ. Jónsson, "Smart contract upgradeability on the ethereum blockchain platform: An exploratory study," *arXiv preprint arXiv:2304.06568*, 2023.
[27] T. Durieux, J. Ferreira, R. Abreu, and P. Cruz, "Smartbugs: A framework to analyze solidity smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1341–1344, IEEE, 2020.
[28] M. Di Angelo, T. Durieux, J. F. Ferreira, and G. Salzer, "Smartbugs 2.0: An execution framework for weakness detection in ethereum smart contracts," in *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 2102–2105, IEEE, 2023.
[29] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, pp. 530–541, 2020.
[30] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools?," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 415–427, ACM, 2020.
[31] C. Torres, D. Ferreiro, B. Livshits, and P. Rümmer, "Are we there yet? unraveling the state-of-the-art smart contract fuzzers," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 329–340, ACM, 2021.
[32] B. Dia, N. Ivaki, and N. Laranjeiro, "An empirical evaluation of the effectiveness of smart contract verification tools," in *Proceedings of the 2021 IEEE 26th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pp. 17–26, IEEE, 2021.
[33] Decurity, "Semgrep rules for smart contracts." https://github.com/Decurity/semgrep-smart-contracts. Accessed: 2024-08-24.
[34] Protofire, "Solhint." https://github.com/protofire/solhint. Accessed: 2024-08-24.
[35] Torres, Christof, "Honeybadger." https://github.com/christoftorres/HoneyBadger. Accessed: 2024-08-24.

[36] Melonport AG, "Conkas." https://github.com/melonproject/oyente/tree/maian. Accessed: 2024-08-24.
[37] Trail of Bits, "Manticore." https://github.com/trailofbits/manticore. Accessed: 2024-08-24.
[38] Torres, Christof, "Osiris." https://github.com/christoftorres/Osiris. Accessed: 2024-08-24.
[39] "Smart contract weakness classification registry," 2024. Accessed: Aug. 2024. [Online]. Available: https://swcregistry.io/.
[40] L. S. H. Colin, P. M. Mohan, J. Pan, and P. L. K. Keong, "An integrated smart contract vulnerability detection tool using multi-layer perceptron on real-time solidity smart contracts," *IEEE Access*, 2024.
[41] N. Songsom, W. Werapun, J. Suaboot, and N. Rattanavipanon, "The swc-based security analysis tool for smart contract vulnerability detection," in *2022 International Conference on Information Technology*, 2022.
[42] Z. Zheng, J. Su, J. Chen, D. Lo, Z. Zhong, and M. Ye, "Dappscan: Building large-scale datasets for smart contract weaknesses in dapp projects," *IEEE Transactions on Software Engineering*, 2024.
[43] P. Zhang, F. Xiao, and X. Luo, "A framework and dataset for bugs in ethereum smart contracts," in *2020 IEEE international conference on software maintenance and evolution (ICSME)*, pp. 139–150, IEEE, 2020.
[44] C. S. Yashavant, S. Kumar, and A. Karkare, "Scrawld: A dataset of real world ethereum smart contracts labelled with vulnerabilities," *arXiv preprint arXiv:2202.11409*, 2022.
[45] M. Soud, I. Qasse, G. Liebel, and M. Hamdaqa, "Automesc: Automatic framework for mining and classifying ethereum smart contract vulnerabilities and their fixes," in *2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 410–417, IEEE, 2023.
[46] Z. Zhang, B. Zhang, W. Xu, and Z. Lin, "Demystifying exploitable bugs in smart contracts," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 615–627, IEEE, 2023.
[47] D. Perez and B. Livshits, "Smart contract vulnerabilities: Vulnerable does not imply exploited," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1325–1341, 2021.
[48] M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, C. Sun, H. Li, and Y. Cai, "Empirical evaluation of smart contract testing: What is the best choice?," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 566–579, 2021.
[49] Z. Zheng *et al.*, "Dappscan: Building large-scale datasets for smart contract weaknesses in dapp projects," *IEEE Transactions on Software Engineering*, 2024.
[50] J. Cai *et al.*, "Fine-grained smart contract vulnerability detection by heterogeneous code feature learning and automated dataset construction," *Journal of Systems and Software*, vol. 209, p. 111919, 2024.
[51] V. Bui, S. Wen, J. Yu, and X. Xia, "Evaluating upgradable smart contract," *Journal of Software Engineering and Applications*, 2021.
[52] N. Girish and S. Kaganurmath, "Upgradability of smart contracts: A review," *International Journal of Recent Engineering Research and Development*, 2022.
[53] J. Ferreira, "Specification is law: Safe creation and upgrade of ethereum smart contracts," in *International Conference on Blockchain and Distributed Ledger Technology*, 2022.
[54] J. L. Marin, "Upgradeable smart contracts design patterns for dapps architectures," in *International Conference on Smart Contracts and DApps*, 2022.
[55] M. Salehi, "An analysis of upgradeability, oracles, and stablecoins in the ethereum blockchain," *Journal of Blockchain Technology*, 2022.
[56] I. Qasse, M. Hamdaqa, and B. Þ. Jónsson, "Smart contract upgradeability on the ethereum blockchain platform: An exploratory study," *Blockchain: Research and Applications*, 2023.
[57] W. B. III, S. Meisami, and Y. Duan, "Proxy hunting: Understanding and characterizing proxy-based upgradeable smart contracts in blockchains," in *Proceedings of the 32nd USENIX Security Symposium*, 2023.
[58] A. Ebrahimi, B. Adams, G. Oliva, and A. Hassan, "Upc sentinel: An accurate approach for detecting upgradeability proxy contracts in ethereum," in *Proceedings of the 35th IEEE International Conference on Blockchain Technology*, 2023.
[59] Y. Huang, X. Wu, Q. Wang, Z. Qian, and X. Chen, "The sword of damocles: Upgradeable smart contract in ethereum," *Journal of Blockchain and Cryptography*, 2024.
[60] Y. Liu, S. Li, X. Wu, Y. Li, Z. Chen, and D. Lo, "Demystifying the characteristics for smart contract upgrades," *arXiv preprint arXiv:2406.05712*, 2024.
[61] X. Li, J. Yang, J. Chen, Y. Tang, and X. Gao, "Characterizing ethereum upgradable smart contracts and their security implications," *Journal of Blockchain Security*, 2024.
[62] J. Oberheide, E. Cooke, and F. Jahanian, "If it ain't broke, don't fix it: Challenges and new directions for inferring the impact of software patches," in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2009.
[63] S. Treetippayaruk and T. Senivongse, "Security vulnerability assessment for software version upgrade," in *2017 18th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pp. 153–157, IEEE, 2017.
[64] N. Dissanayake, K. Theagarajan, Y. Liu, R. Kannavara, H. Gonzalez, and M. Piccardi, "Software security patch management—a systematic literature review of challenges, approaches, tools, and practices," *Information and Software Technology*, vol. 144, p. 106771, 2022.
[65] R. Wash, E. Rader, and K. E. Vaniea, "Out of the loop: How automated software updates cause unintended security consequences," in *10th Symposium On Usable Privacy and Security (SOUPS)*, 2014.
[66] K. E. Vaniea, E. Rader, and R. Wash, "Betrayed by updates: How negative experiences affect future security," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 2671–2674, ACM, 2014.
[67] T. Dumitraş, P. Narasimhan, and E. Tilevich, "To upgrade or not to upgrade: Impact of online upgrades across multiple administrative domains," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pp. 865–876, ACM,

2010.

Chapter 7

EvoChain: A Framework for Tracking and Visualizing Smart Contract Evolution

Authors: Ilham Qasse, Mohammad Hamdaqa, and Björn Þór Jónsson
Status: Submitted to the Data and Tool Showcase Track of the 22nd International Conference on Mining Software Repositories (MSR 2025).

Highlights

- Presents EvoChain, a novel framework for tracking and analyzing the evolution of Ethereum smart contracts and their associated proxy relationships.
- Combines graphical and tabular views to provide an interactive interface for exploring contract versions, upgrade paths, and associated vulnerabilities.
- Enables real-time retrieval of verified source code from Etherscan, ensuring access to the latest data for analysis.
- Supports advanced querying capabilities, allowing users to search contracts by address, proxy type, or version details.
- Facilitates empirical studies on smart contract upgradeability by providing comprehensive metadata and visualization tools.

EvoChain: A Framework for Tracking and Visualizing Smart Contract Evolution

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

***Abstract*—Tracking the evolution of smart contracts is challenging due to their immutable nature and complex upgrade mechanisms. We introduce EvoChain, a comprehensive framework and dataset designed to track and visualize smart contract evolution. Building upon data from our previous empirical study, EvoChain models contract relationships using a Neo4j graph database and provides an interactive web interface for exploration. The framework consists of a data layer, an API layer, and a user interface layer. EvoChain allows stakeholders to analyze contract histories, upgrade paths, and associated vulnerabilities by leveraging these components. Our dataset encompasses approximately 1.3 million upgradeable proxies and nearly 15,000 historical versions, enhancing transparency and trust in blockchain ecosystems by providing an accessible platform for understanding smart contract evolution.**

***Index Terms*—Proxy Contracts, Smart Contracts, Immutability, Software Maintenance, Versions**

I. Introduction

Software evolution is a critical aspect of software development, enabling continuous improvement, adaptation to new requirements, and rectification of defects over time [1], [2]. Tracking software versions is essential for developers and stakeholders to understand changes, maintain compatibility, and ensure security throughout the software lifecycle [3]–[5]. In traditional software engineering, version control systems, and software repositories have long facilitated this process [6].

In blockchain technology and smart contracts, tracking software evolution introduces unique challenges. Smart contracts are self-executing code deployed on immutable blockchain platforms such as Ethereum [7]–[9]. Once deployed, their code cannot be altered or upgraded in the conventional sense [10], [11]. To enable updates, developers employ design patterns such as proxy contracts, which delegate calls to upgradeable logic contracts [12]–[14]. However, comprehending a smart contract's history, version dependencies, and associated vulnerabilities remains difficult due to the decentralized and transparent yet inherently complex nature of blockchain systems [15]–[18].

The absence of comprehensive tools and datasets for tracking the evolution of smart contracts creates significant challenges. Developers and auditors lack efficient means to trace modifications across different contract versions, limiting their ability to monitor version changes. Additionally, assessing the impacts of upgrades implemented via proxy mechanisms on functionality and security is restricted. Finally, blockchain data's scattered and complex nature limits identifying and tracing the resolution of security vulnerabilities over time. These challenges impede efforts to ensure reliability, security, and transparency in smart contract ecosystems, ultimately affecting trust and adoption of blockchain technologies.

To address these challenges, we introduce EvoChain, a novel framework designed to track and visualize the evolution of smart contracts. EvoChain integrates:

- A comprehensive dataset, aggregating historical smart contract data, including code versions, deployment transactions, proxy relationships, and known vulnerabilities.
- Graph-based modeling, utilizing Neo4j to represent complex relationships between contracts, proxies, versions, and associated issues in a connected graph structure.
- An interactive visualization tool, offering a user-friendly web application that enables stakeholders to explore contract histories and dependencies without writing complex queries.

EvoChain offers significant contributions to the blockchain and software engineering communities. First, it enhances the understanding of smart contract evolution by systematically capturing and modeling contract versions and their relationships, aiding comprehension of their progression over time. Second, EvoChain facilitates security analysis by enabling the identification and tracing of vulnerabilities across contract versions, supporting efforts to improve contract security. Finally, it improves accessibility and transparency through an intuitive web interface that lowers the barrier for developers, auditors, and researchers to analyze smart contract evolution, promoting trust in decentralized applications. By leveraging EvoChain, stakeholders can analyze smart contract upgrades, trace vulnerabilities, and facilitate greater transparency in blockchain ecosystems.

II. EvoChain Overview

EvoChain is a modular framework designed to track and visualize the evolution of smart contracts. EvoChain provides actionable insights into contract histories, upgrade motivations, and associated vulnerabilities by leveraging data from a prior empirical study, integrating graph-based modeling, and an interactive user interface. This section presents an overview of the architecture, encompassing the data layer, API layer, and user interface while detailing the methodology and its connection to prior work.

A. Data Layer

The data layer of EvoChain provides the foundation for tracking smart contract evolution. In this section, we discuss the data sources, the methodology used to collect and process the data, and the schema and storage mechanism employed to organize and query the dataset.

1) Data Sources: EvoChain builds upon a dataset derived from our previous empirical study [13] which analyzed the lifecycle of upgradeable smart contracts on Ethereum. The two primary data sources for EvoChain are:

- EthereumETL[1]: An open-source tool providing detailed historical data from the Ethereum blockchain, including blocks, transactions, logs, and events. This dataset serves as the foundation for analyzing interactions and upgrading patterns of smart contracts.
- Etherscan API[2]: A widely used Ethereum block explorer providing verified smart contract source code, metadata, and additional details like creation timestamps and transaction counts. Etherscan complements the EthereumETL dataset by supplying off-chain information critical for code analysis and validation.

2) Methodology Process: The data collection process involves the following key steps:

1) Filtering Upgradeable Contracts: In the study we employed PROXIFY,[3] a tool designed to detect upgradeable smart contracts. PROXIFY analyzes contract bytecode and emitted events for patterns indicative of proxy usage, such as delegatecall operations and specific storage arrangements.
2) Tracing Historical Versions:
   - By examining emitted events (e.g., Upgraded, ImplementationUpdated) and transaction logs, EvoChain maps each proxy contract to the various implementation contracts it has managed over time, forming a complete upgrade history.
   - Events containing addresses of new implementations are analyzed to reconstruct the sequence of versions associated with each proxy, revealing the relationships between proxies and their implementations.
3) Fetching Source Code and Metadata:
   - Using the Etherscan API, EvoChain retrieves the verified source code for each contract version.
   - Additional metadata, including the contract's creation timestamp, transaction count, is gathered to contextualize each version.
4) Observed Changes in Smart Contracts: In our previous study [13], we categorized observed changes into four types: fixing vulnerabilities (addressing security issues), feature modifications (altering functionality through additions or deletions), gas optimizations (reducing gas costs for performance improvements), and other changes (minor adjustments or policy updates not fitting the other categories). We utilize tools such as Git diff[4] to compare code differences between versions and the SmartBugs frameworkk[5] to detect security vulnerabilities. Gas cost comparisons are made using actual deployment data from Etherscan, providing a reliable overview of structural optimizations.

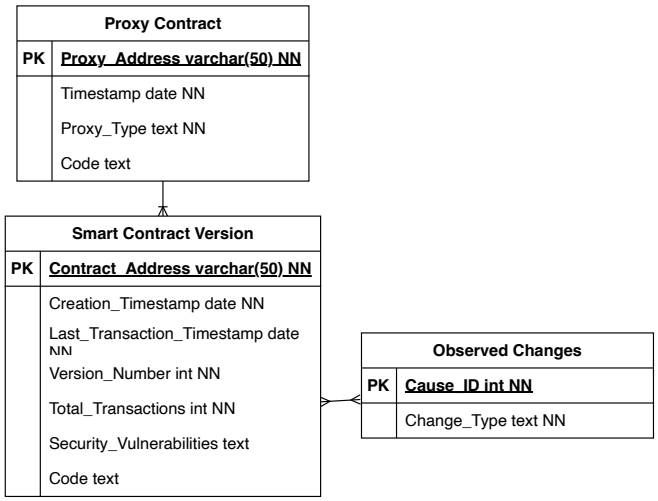

Fig. 1: EvoChain Data Schema

3) Data Schema: The dataset is structured as a graph in Neo4J, capturing the relationships between smart contract versions, proxies, and their upgrade paths. Core entities include:

- Smart Contract Versions: Nodes representing contract versions, with attributes such as contract address, creation timestamp, last transaction timestamp, total transactions, version number, and vulnerabilities.
- Proxy Contracts: Nodes representing proxies, classified by type (e.g., EIP-1967, Transparent Proxy) and linked to the contracts they manage.

Key relationships include:

- Implements: A one-to-many relationship linking a proxy contract to the smart contract versions it controls.
- Observed Changes: Annotates the nature of the change introduced in each version (e.g., security fix, performance optimization, feature addition).

Figure 1 illustrates the graph schema, depicting the nodes, attributes, and relationships central to EvoChain.

B. API Layer

The API layer, built with Flask, serves as an intermediary between the data layer and the user interface, enabling seamless user interaction with the underlying data. Its key functions include:

[1]https://ethereum-etl.readthedocs.io/en/latest/
[2]https://etherscan.io
[3]https://github.com/IlhamQasse/PROXiFY
[4]https://git-scm.com/docs/git-diff
[5]https://hub.docker.com/u/smartbugs

- Query Execution: Processes user inputs to retrieve data from the Neo4j database, such as contract relationships, version histories, and security information.
- Dynamic Data Retrieval: Integrates with the Etherscan API for real-time retrieval of verified source code and metadata, ensuring users have access to the most up-to-date information.
- Data Transformation: Structures raw query results into user-friendly formats suitable for visualization in the user interface.

This modular API design abstracts the complexity of interacting with graph databases and blockchain data, allowing users to focus on insights rather than implementation details.

C. User Interface Layer

The user interface provides an interactive platform for exploring the evolution of smart contracts, combining graphical and tabular views for comprehensive analysis. Key features include:

- Graphical Visualization: EvoChain visualizes contract versions and proxy relationships as nodes and edges. Users can interact with the graph to trace upgrade paths, explore vulnerabilities, and identify root causes of changes.
- Tabular View: A synchronized tabular display lists detailed metadata for selected nodes, including creation timestamps, transaction counts, and vulnerabilities.
- Querying Capabilities: Users can search for contracts by address, proxy type, or version details, enabling flexible and targeted analysis.
- Real-Time Code Retrieval: The interface dynamically fetches verified source code from Etherscan, ensuring users can access the latest data.

The EvoChain dataset and tool are publicly available online[6] to support research and exploration of smart contract evolution.

III. Applications of EvoChain

EvoChain bridges a critical gap in the blockchain ecosystem by bringing the principles of version tracking and transparency, which are long established in traditional software development, to smart contracts. In conventional software engineering, version control systems like Git enable developers to track changes, collaborate effectively, and maintain a history of modifications [1]. EvoChain extends these capabilities to smart contracts, which traditionally lack such comprehensive evolution tracking due to the immutable nature of blockchain deployments.

One significant application of EvoChain is in enhancing transparency and user trust. By providing a clear and accessible history of smart contract versions, upgrades, and associated changes, users can make informed decisions about which contracts to interact with. This level of transparency is not typically available through standard blockchain explorers or tools, which often do not provide detailed insights into a contract's evolution or the reasons behind upgrades. EvoChain's visualization of upgrade paths and observed changes empowers users with knowledge about the contract's reliability and the responsiveness of developers to issues such as security vulnerabilities.

In security auditing and vulnerability analysis, EvoChain is a powerful tool for auditors and security professionals. By examining the relationships between contract versions and their associated vulnerabilities, analysts can track the remediation of known security issues over time. This continuous monitoring facilitates comprehensive risk assessments and supports the verification of adherence to security standards throughout a contract's lifecycle. EvoChain's ability to highlight unresolved vulnerabilities and the effectiveness of past upgrades enhances the efficiency of auditing processes and aids in proactive risk management.

For developers, EvoChain offers valuable insights into smart contract development and maintenance practices. Developers can learn from previous modifications by exploring upgrade paths and analyzing observed changes(such as bug fixes, feature additions, or gas optimizations). This knowledge helps in planning future upgrades more effectively, avoiding past mistakes, and adopting best practices. EvoChain's visualization of contract evolution aids in understanding the impact of changes, facilitating more strategic decision-making in the development process.

EvoChain also has significant applications in academic research and machine learning. Researchers can leverage the rich dataset provided by EvoChain to conduct large-scale empirical studies on upgrade patterns, security trends, and maintenance activities in smart contracts. This data can be instrumental in training machine learning models for various purposes, such as predicting potential vulnerabilities, assessing the risk of future upgrades, or forecasting contract evolution. By applying techniques like anomaly detection and pattern recognition, researchers can develop predictive analytics tools that enhance smart contracts' security and reliability.

Finally, in the context of investment decision-making, EvoChain enables investors and stakeholders to analyze the historical evolution of smart contracts. The transparency provided by the tool increases confidence in the reliability and integrity of contracts. Investors can estimate their potential investments' activity, longevity, and safety by assessing total transactions, contract age, and the history of security assessments. This informed approach aids in identifying trustworthy contracts and understanding the risks associated with interacting with specific smart contracts.

IV. Limitations and Future Work

While EvoChain provides significant advancements in tracking smart contract evolution, certain limitations present opportunities for future enhancement. One of the primary limitations is the reliance on emitted events to detect upgrades. Since EvoChain depends on standardized upgrade events emitted by smart contracts, those that do not emit such events are not

[6]https://github.com/IlhamQasse/EvoChain.git

fully captured, leading to incomplete data. This reliance may overlook upgrades performed without event emissions or using unconventional methods, affecting the comprehensiveness of the dataset.

Another limitation is the focus on proxy-based upgrade patterns. While proxies are prevalent in enabling smart contract upgrades, they are not the only mechanism. EvoChain currently does not extensively analyze other upgrade approaches, such as data separation or strategy pattern. This narrow focus may introduce bias and limit insights into alternative upgrade practices within the blockchain ecosystem. EvoChain's scope is also currently limited to the Ethereum blockchain, excluding contracts deployed on other platforms like Binance Smart Chain or Polkadot. This Ethereum-centric approach restricts the ability to compare practices and trends across different blockchain environments, potentially overlooking unique evolution patterns present in other ecosystems.

Scalability challenges present another limitation. Processing and storing the vast amount of data from the Ethereum blockchain poses difficulties in terms of data volume management and performance constraints. Maintaining responsive query performance and efficient data retrieval becomes increasingly complex as the dataset grows.

To address these limitations, future work on EvoChain will focus on several enhancements. One key improvement is the development of enhanced data collection methods. By incorporating alternative detection techniques, such as function call analysis or code similarity detection, EvoChain can identify upgrades beyond emitted events. This approach aims to capture a wider range of upgrade practices, improving the completeness of the dataset.

Another area of focus is promoting standardization within the developer community. By advocating for the adoption of standardized event naming conventions and logging practices, EvoChain can enhance the reliability of data collection and facilitate more accurate tracking of contract evolution. Collaboration with industry stakeholders and participation in standardization initiatives can drive this effort.

Expanding EvoChain's capabilities to include multiple blockchain platforms is another important direction for future work. Cross-platform support will enable comparative analyses and broaden the tool's applicability. Investigating how contracts evolve across different platforms will provide insights into best practices and common challenges, enriching the understanding of smart contract evolution globally.

Addressing scalability challenges involves implementing optimized data storage solutions and performance enhancements. Employing scalable architectures, database optimization techniques, and efficient caching strategies can manage large datasets effectively while maintaining responsive user interactions. These technical improvements will ensure that EvoChain remains a robust and user-friendly tool as it grows.

Lastly, including alternative upgrade mechanisms in the analysis will provide a more holistic view of smart contract evolution. By extending the scope beyond proxy patterns, EvoChain can analyze various upgrade methods, supporting more comprehensive insights and use cases.

By addressing these limitations and pursuing these enhancements, EvoChain aims to evolve into a more robust and versatile tool. These improvements will strengthen EvoChain's role in facilitating secure, transparent, and efficient smart contract development and maintenance, ultimately contributing to the advancement of blockchain technology.

V. Related Work

The detection and analysis of upgradeable proxy contracts have been the focus of several studies, employing methods such as source code analysis [19], [20], bytecode analysis [21], [22], and transaction history [12], [23]. While these studies advanced the understanding of proxy contracts, they primarily concentrated on detecting active proxies or their upgradeability features without focusing on comprehensively tracking historical versions. Only a few studies have explored the evolution of smart contracts. Li et al. [22] and Liu et al. [20] provided limited insights into historical versions, detecting 4,692 and 973 versions, respectively, but lacked a comprehensive view of changes over time, as summarized in Table I. In contrast, EvoChain surpasses these limitations by offering a comprehensive dataset and tool for tracking the evolution of smart contracts. It traces 14,990 historical versions across 1.3 million upgradeable proxies, significantly outscoring prior studies. Furthermore, EvoChain uniquely provides an interactive visualization tool, enabling users to explore smart contract evolution in depth. Unlike previous studies, EvoChain also makes its dataset publicly available, facilitating further research and enhancing transparency in the blockchain ecosystem.

TABLE I: Summary of Studies on Upgradeable Proxy Contracts and Their Evolution

Study	# Proxies Detected	# Versions Tracked
[12]	8,225	N/A
[19]	8,815	N/A
[23]	∼3,000	N/A
[20]	44,282 (total)	973
[22]	43,650	4,692
EvoChain	∼1,300,000	14,990

VI. Conclusion

We presented EvoChain, a framework and dataset for tracking and visualizing smart contract evolution on Ethereum. By leveraging data from our previous study, modeling it in a Neo4j graph database, and providing an interactive web interface, EvoChain addresses challenges in understanding contract upgrades and vulnerabilities. The tool facilitates applications in security auditing, development insights, academic research, and investment decision-making. Despite limitations such as reliance on emitted events and focus on proxy patterns, EvoChain significantly enhances transparency and trust in blockchain ecosystems. Future work includes expanding to

other blockchains, integrating predictive analytics, and improving data collection methods to capture a broader range of upgrade practices.

REFERENCES

[1] T. Mens, S. Demeyer, and T. Mens, *Introduction and roadmap: History and challenges of software evolution*. Springer, 2008.

[2] M. M. Lehman and J. F. Ramil, "Software evolution and software evolution processes," *Annals of Software Engineering*, vol. 14, pp. 275–309, 2002.

[3] D. Spinellis, "Version control systems," *IEEE Software*, vol. 22, no. 5, pp. 108–109, 2005.

[4] R. Conradi and B. Westfechtel, "Version models for software configuration management," *ACM Computing Surveys*, vol. 30, no. 2, pp. 232–282, 1998.

[5] J. Estublier, "Software configuration management: A roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, pp. 279–289, ACM, 2000.

[6] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 3, pp. 309–346, 2002.

[7] N. Szabo, "Smart contracts: Building blocks for digital markets," *Extropy*, no. 16, 1996.

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

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

[10] M. Hamdaqa, L. A. P. Met, and I. Qasse, "icontractml 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms," *Information and Software Technology*, vol. 144, p. 106762, 2022.

[11] B. K. Mohanta, S. S. Panda, and D. Jena, "An overview of smart contract and use cases in blockchain technology," in *2018 9th international conference on computing, communication and networking technologies (ICCCNT)*, pp. 1–4, IEEE, 2018.

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

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

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

[15] W. Jiang, Y. Zhang, H. Lei, *et al.*, "Contractward: Automated smart contract vulnerability detection based on business process mining," *Computers & Security*, vol. 112, p. 102502, 2022.

[16] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Proceedings of the 6th International Conference on Principles of Security and Trust (POST)*, pp. 164–186, Springer, 2017.

[17] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 254–269, ACM, 2016.

[18] M. Soud, I. Qasse, G. Liebel, and M. Hamdaqa, "Automesc: Automatic framework for mining and classifying ethereum smart contract vulnerabilities and their fixes," in *2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 410–417, IEEE, 2023.

[19] W. E. Bodell III, S. Meisami, and Y. Duan, "Proxy hunting: Understanding and characterizing proxy-based upgradeable smart contracts in blockchains," in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 1829–1846, 2023.

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

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

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

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

Chapter 8

Conclusion

This thesis has delivered a comprehensive analysis of smart contract upgradeability patterns by establishing standardized definitions, documenting practical methods, and examining real-world adoption and security impacts. Building upon this analysis, we developed robust tools and frameworks that provide developers and researchers with the necessary guidelines to effectively manage the complexities of upgrading smart contracts while maintaining security and user trust. In this chapter, we summarize the key results and contributions of this thesis, highlighting their significance for smart contract development and management. This is followed by an in-depth discussion that revisits the main question posed in the introduction: to upgrade or not? The chapter then reflects on the limitations of this thesis and examines its broader implications for the smart contract ecosystem. Finally, we conclude with open questions and final remarks, offering directions for future research and emphasizing the transformative potential of secure and adaptable upgrade mechanisms.

8.1 Summary of Results

The research was structured around five core research objectives (RO1–RO5), each addressing specific research gaps (G1–G5). Through this work, we systematically investigated smart contract upgradeability to enhance security, transparency, and reliability within blockchain applications, specifically focusing on Ethereum due to its open-source architecture, widespread adoption, and active community support. Figure 8.1 illustrates the relationships between the research gaps, objectives, results, and contributions, providing an integrative summary of how each research component aligns to achieve the overall aim.

8.1.1 RO1: Systematic Classification and Analysis of Smart Contract Upgrade Approaches

To address the absence of standardized definitions and classifications in smart contract upgrade mechanisms (research gap G1), we conducted a Multivocal Literature Review (MLR) that established a structured taxonomy and characterization of upgrade approaches, addressing three research questions:

- **RQ1.1: What are the existing approaches for upgrading Ethereum smart contracts?**

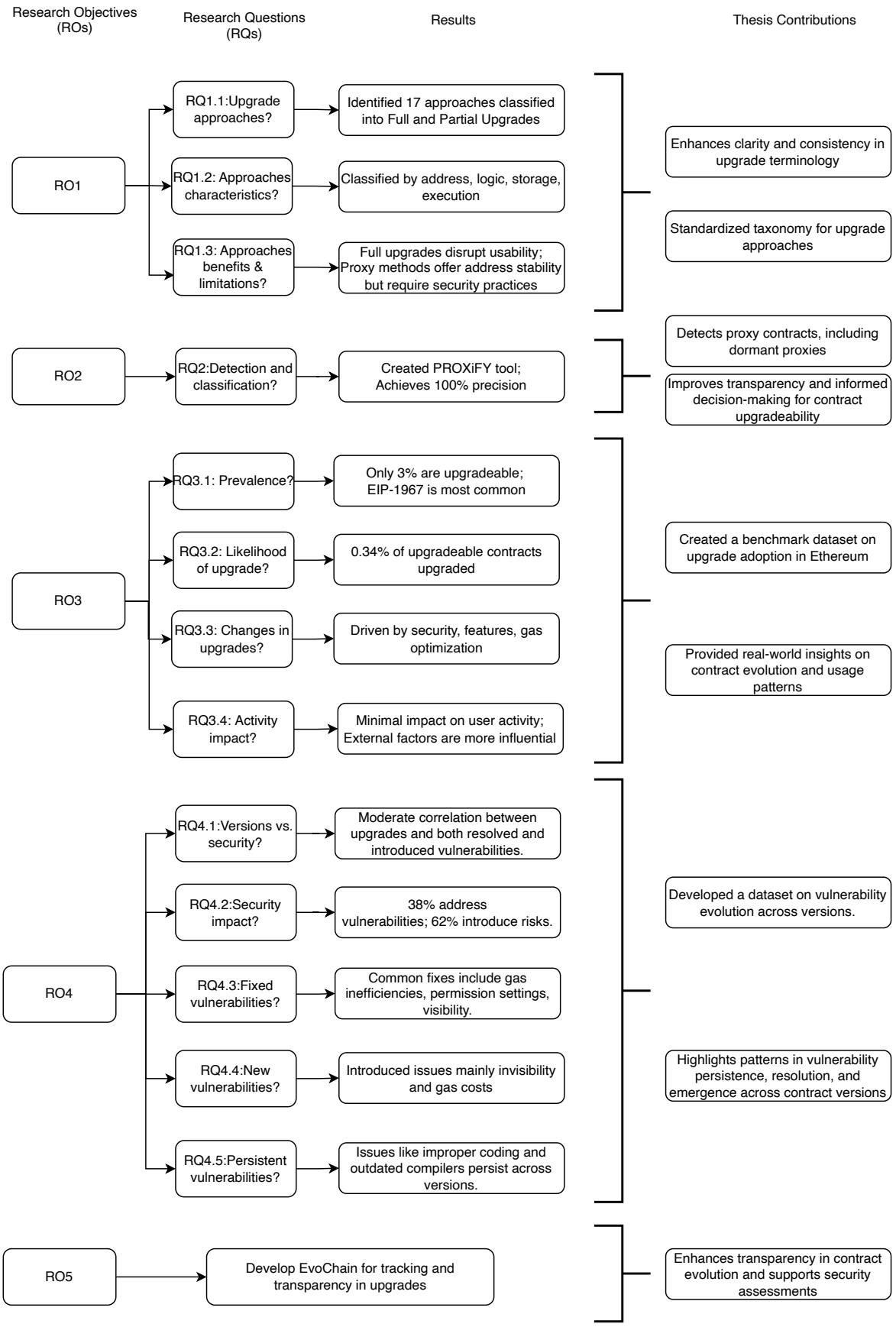

Figure 8.1: Summary of research objectives, results, and thesis contributions

Through a detailed review, we identified two primary categories of upgrade mechanisms: full upgrades and partial upgrades. Full upgrades include **Contract Migration** and **Metamorphic Contracts**, which enable complete replacement of the contract while requiring new addresses for user interaction, potentially disrupting usability. Partial upgrades include **Proxy Patterns** (such as Basic Proxy, Transparent Proxy, and UUPS Proxy), the **Diamond Standard (EIP-2535)**, **Data Separation** techniques (Unstructured Storage, Eternal Storage), and **Hybrid Approaches** that combine proxy mechanisms with data separation. This classification provides a consistent terminology and organizational framework for understanding upgrade mechanisms.

- **RQ1.2: What are the characteristics of each upgrading approach?**

  We classified upgrade approaches based on address preservation, logic upgrade scope, storage management, and execution flow. Proxy-based, Multi-Module and Hybrid approaches generally preserve a fixed address, while Contract Migration and Data Separation approach use a new address, impacting continuity. Full logic replacement is supported by Proxy-based, Hybrid, and Data Separation approaches, whereas Multi-Module approaches enable modular, partial replacement. For storage, Proxy-based and Multi-Module approaches maintain state, while Contract Migration and Metamorphic Contracts require state migration. Finally, Proxy-based approaches use delegate calls for flexible logic updates without altering state, while Contract Migration and direct-call methods, such as Strategy Pattern, offer simpler but less modular execution.

- **RQ1.3: What are the benefits and limitations of each approach?**

  The benefits and limitations vary significantly across upgrade mechanisms. **Contract Migration**, while straightforward, reduces usability as it necessitates new contract addresses, disrupting continuity for users. **Proxy Patterns** maintain consistent addresses and offer flexibility, but they depend on careful security practices due to potential vulnerabilities from `delegatecall`. The **Diamond Standard** offers extensive modularity and flexibility, ideal for complex systems, though it demands a high level of expertise to manage securely. **Hybrid Approaches** leverage the strengths of proxy and data separation techniques, providing moderate flexibility and reduced complexity. These findings fill the gap in definitions and classifications, offering standardized insights that improve communication and best practices across the blockchain community.

As part of this objective, we established a unified definition of upgradeability in smart contracts. Upgradeability in smart contracts refers to the capability to modify the contract's logic or code post-deployment. This modification can occur with or without preserving the contract's state and may involve full or partial changes to the contract's components. This definition aligns both academic and industry perspectives, addressing inconsistencies in terminology and understanding.

8.1.2 RO2: Developing a Tool for Detecting Proxy-Based Upgradeable Contracts

To address the user challenges in identifying upgradeable contracts (research gap G2), we developed **PROXiFY**, a bytecode-based detection tool designed to identify proxy-

based upgradeable contracts accurately. This research objective addressed the following question:

- **RQ2: How can proxy-based upgradeable contracts be accurately detected and classified?** PROXiFY analyzes smart contract bytecode to detect `delegatecall` instructions and traces implementation addresses to identify and classify different proxy patterns without requiring source code or transaction data. Testing showed that PROXiFY enhances detection accuracy by up to 100% over previous methods, particularly in identifying dormant proxies that were previously undetectable. This tool allows users, developers, and auditors to reliably assess a contract's upgradeability status, empowering them to make informed choices.

8.1.3 RO3: Empirical Analysis of Proxy-Based Upgrade Patterns in Ethereum

To understand the real-world prevalence and application of upgradeable contracts (research gap G3), we conducted an empirical study of approximately 44 million Ethereum contracts, examining four key questions:

- **RQ3.1: How prevalent are proxy-based upgrade patterns in Ethereum smart contracts?**

  Our analysis revealed that around **3%** of the analyzed contracts exhibit upgradeable characteristics, with **EIP-1967 (Proxy Logic Storage Slots)** being the most widely used proxy standard. Despite their presence, these upgradeable contracts remain a small portion of the overall contract landscape, suggesting limited adoption.

- **RQ3.2: How likely is an upgradeable contract to be upgraded?**

  Of the upgradeable contracts identified, only a small fraction (approximately **0.34%**) were actually upgraded post-deployment. Most contracts maintain their initial version, reflecting a preference for stability among developers who may avoid upgrades due to potential risks or complexities.

- **RQ3.3: What changes occur in smart contracts post-upgrade?**

  When upgrades do occur, they are generally driven by the need to address security vulnerabilities, add features, or optimize gas usage. Complex upgrades that modify multiple aspects of a contract are rare due to the challenges involved. This cautious approach highlights a prioritization of essential updates, focusing on security and performance rather than extensive modifications.

- **RQ3.4: How does upgrading impact the activity level of smart contracts?**

  The study showed a nuanced relationship between upgrades and activity levels, with most upgraded contracts experiencing minimal user activity. A significant portion of contract versions recorded only the initial deployment transaction. Factors such as user behavior and market conditions play a more influential role in activity than the mere presence of upgrades. This empirical data helps tailor upgrade mechanisms to meet real-world needs, supporting the cautious but essential adoption of upgrades.

8.1.4 RO4: Security Impact Analysis of Smart Contract Upgrades

Given the dual potential of upgrades to both enhance and undermine security (research gap G4), we analyzed the security effects of contract upgrades through five research questions:

- **RQ4.1: Is there a correlation between the number of versions a smart contract undergoes and overall security improvement or degradation?**

  Our analysis found a moderate correlation between the number of versions a contract has and the occurrence of both fixed and introduced vulnerabilities. While some upgrades successfully addressed security issues, they also tended to introduce new ones, reflecting the complexity of maintaining security through upgrades.

- **RQ4.2: How do smart contract upgrades impact contract security?**

  Upgrades positively impacted security in about **38%** of cases, addressing known vulnerabilities. However, in **62%** of cases, upgrades either failed to resolve existing issues or introduced new security risks, suggesting that upgrades require comprehensive security assessments to avoid unintended vulnerabilities.

- **RQ4.3: What are the most common vulnerabilities fixed while upgrading smart contracts?**

  Commonly addressed vulnerabilities included gas inefficiencies, permission settings, and visibility issues. These are critical areas that impact contract cost efficiency, security, and correct functionality.

- **RQ4.4: What are the most common vulnerabilities introduced during upgrades?**

  Newly introduced vulnerabilities often mirrored these issues, particularly in function visibility and gas-related costs. The recurrence of similar vulnerabilities suggests that upgrades are not always accompanied by rigorous testing and validation processes.

- **RQ4.5: Do some vulnerabilities persist across versions despite upgrade efforts?**

  Certain vulnerabilities, such as improper coding practices and outdated compiler usage, persisted across multiple contract versions, often due to the complexity of resolving them without significant refactoring. This finding underscores the need for proactive, ongoing security practices in smart contract management, addressing both new and recurrent vulnerabilities through rigorous testing.

8.1.5 RO5: Developing a Framework for Tracking Smart Contract Evolution

To enhance transparency in contract upgrades (research gap G5), we developed **EvoChain**, a framework that allows stakeholders to track contract evolution with visual clarity. EvoChain provides a visual timeline of contract history, allowing developers, auditors, and users to easily access and verify a contract's entire lifecycle, from initial deployment

through each subsequent upgrade. By recording each version's specific modifications and linking them with relevant security evaluations, EvoChain supports an accessible audit trail that promotes trust in contract evolution. The user-friendly interface offers stakeholders a comprehensive view of contract history, making it easier to assess upgrade paths and understand the security implications of changes.

8.2 To Upgrade or Not?

The question of whether to make a smart contract upgradeable is a central theme of this thesis and one of the most debated trade-offs in blockchain design. At its core, this question reflects two opposing arguments: On one hand, upgradeability enhances functionality, adaptability, and security, but on the other, it challenges the immutability and trust that are fundamental to blockchain systems.

Our findings from RO3 provide a nuanced perspective on this choice. While only 3% of contracts were identified as upgradeable proxies, these contracts collectively manage over $3.3 billion in Total Value Locked (TVL), highlighting their significant economic impact. Despite their importance, upgrades are infrequent, reflecting a cautious approach among developers who prioritize stability and user trust, particularly in high-stakes applications like decentralized finance (DeFi).

In this section, we first present our position on the contextual adoption of upgradeability, grounded on this thesis's empirical and theoretical findings. We then identify contexts where upgradeability is advantageous, examine the associated risks, and propose best practices to mitigate these risks while balancing flexibility with security and transparency.

8.2.1 Our Stand: Contextual and Careful Adoption of Upgradeability

Based on these insights, we believe that the decision to make a smart contract upgradeable should be contextual and carefully considered. Upgradeability is not inherently good or bad; rather, its suitability depends on the project's specific needs, risks, and goals.

Our analysis suggests several contexts where upgradeability may be both advantageous and necessary:

- High-Value Applications: In projects where significant financial or operational value is at stake, upgradeability can be crucial for maintaining security and addressing emerging threats. For example, in the Compound V2 case study, multiple upgrades were essential to enhance security and functionality, ensuring the protocol's resilience despite managing large amounts of Total Value Locked (TVL).

- Evolving Requirements: Applications that must adapt over time due to changing user needs, technological advancements, or regulatory changes benefit from the flexibility upgradeability provides. For instance, the Aave protocol [1] has undergone multiple upgrades to introduce new features such as improved collateral management and risk parameters, allowing it to remain competitive while adapting to evolving market demands and re

[1] `https://aave.com/`

- Long-Term Projects: Blockchain projects are generally intended to operate indefinitely. However, those with ambitious goals, significant user bases, or complex functionalities face greater challenges in maintaining long-term sustainability. Upgradeability enables such projects to remain functional and relevant as the ecosystem evolves. Ethereum itself exemplifies this need, as its continuous upgrades, such as the transition to Ethereum 2.0[2], address scalability, security, and energy efficiency to support its widespread adoption and diverse use cases.

8.2.2 Mitigating Risks Associated with Upgradeability

To address the inherent risks of upgradeability, such as undermining the perceived immutability of smart contracts or exposing the system to new vulnerabilities, we advocate for several best practices:

- Adopting decentralized governance, such as multi-signature wallets and token-based voting, can distribute control over upgrades, mitigating risks of unilateral or malicious changes. This approach is essential for maintaining user trust and ensuring that no single entity holds disproportionate influence over the upgrade process.
- Comprehensive testing, formal verification, and meticulous management of upgrade mechanisms (e.g., preventing storage collisions) are critical in reducing technical vulnerabilities. Insights from our Multivocal Literature Review (MLR) on delegatecall-based proxies emphasize the importance of managing storage layout and access to prevent unauthorized actions.
- Open communication with users regarding potential upgrades, including advance notice and details of the changes, is vital for maintaining trust. The Compound V2 case study[3] exemplifies how transparency in upgrade processes can positively impact user interaction and trust. By introducing open governance mechanisms, such as community discussions and on-chain voting for proposed upgrades, Compound ensured that users were actively involved in decision-making processes. This approach not only increased user engagement but also fostered greater confidence in the protocol, as stakeholders were assured that upgrades aligned with community interests. Such practices demonstrate the effectiveness of transparency in preserving user confidence, particularly in high-stakes decentralized finance (DeFi) applications.
- Where feasible, providing users with an opt-in upgrade option allows them to adopt upgrades voluntarily. This balances flexibility with users' expectations of immutability, ensuring that functional enhancements do not compromise user autonomy.

In conclusion, we advocate a balanced approach to upgradeability. Rather than universal adoption or avoidance, developers should assess the relevance of upgradeability based on project-specific needs, implementing it thoughtfully and prioritizing user trust. This approach enables blockchain applications to harness the benefits of upgradeability while respecting the fundamental principles of immutability and transparency.

[2] https://ethereum.org/en/upgrades/
[3] https://compound.finance/

8.3 Limitations and Considerations

While our findings provide valuable insights, several limitations of this study should be acknowledged. Firstly, the focus on Ethereum as the primary platform limits the generalizability of our findings across other blockchain ecosystems. Although Ethereum's open-source architecture and widespread adoption make it an ideal case study, the findings may not fully apply to blockchains with distinct consensus mechanisms or programming models, such as Hyperledger Fabric or Cardano. Future research should explore upgrade mechanisms in other platforms to provide a more holistic view of upgrade practices.

Secondly, while PROXiFY effectively identifies proxy-based upgradeable contracts within Ethereum, its focus on established upgrade patterns may limit its applicability to non-standard or newer upgrade mechanisms. Additionally, as upgrade mechanisms evolve, PROXiFY's detection algorithms will require periodic updates to maintain accuracy. Extending PROXiFY's capabilities across blockchain platforms and upgrade patterns would broaden its utility and relevance.

The smart contract versions dataset generated in RO3 and used in EvoChain also introduces certain limitations. It relies on standardized events emitted by smart contracts to detect upgrades, meaning contracts that do not emit such events or use unconventional upgrade methods may not be fully captured. This dependency can result in incomplete datasets, potentially overlooking important upgrade activities.

In RO3, analyzing the types of changes made to smart contracts after upgrades presented several challenges. The categorization of changes relied on observable modifications in the code and did not account for the developer's intent or the underlying rationale behind each change. For example, a security fix might also optimize gas usage, but static analysis alone cannot determine the primary purpose of such modifications. Additionally, minor or ambiguous changes that did not clearly align with predefined categories, such as security, features, or gas optimizations, were grouped under a general "other" category. While this approach is practical, it risks overlooking broader implications, such as policy adjustments or subtle bug fixes. Addressing these challenges would require more granular techniques or supplementary context, such as reviewing developer documentation or communications.

The analysis of vulnerabilities associated with upgrade mechanisms relied on historical contract data. This approach, while valuable, may not capture emerging risks, novel attack vectors, or evolving best practices. Incorporating real-time monitoring, adaptive threat detection, and predictive analytics into upgrade assessment tools could address this limitation, enabling proactive identification and mitigation of vulnerabilities.

Finally, this thesis primarily focused on technical aspects, with limited attention to socio-technical factors such as developer experience, community governance, and organizational policies that shape upgrade practices. A more comprehensive understanding of these human and organizational dynamics would offer a broader perspective on the impact of upgradeability within the blockchain ecosystem.

8.4 Broader Implications and Practical Applications

The findings of this research have significant implications for blockchain technology, particularly in enhancing user trust, transparency, and security. Secure upgradeability

is essential for applications in high-stakes sectors such as decentralized finance (DeFi), supply chain management, and healthcare, where contracts must adapt to regulatory changes and evolving user needs. By providing a verifiable record of contract evolution through tools like EvoChain, this research contributes to a framework that supports transparency and trust.

The practical tools developed in this thesis, including PROXiFY and EvoChain, directly address these challenges. PROXiFY enhances transparency by enabling users to assess a contract's upgradeability status, which is critical in preventing malicious exploitation of hidden upgrade paths post-deployment. Similarly, EvoChain supports regulatory compliance and governance by providing a transparent record of contract history, empowering stakeholders to track upgrades and verify contract integrity across various industries.

Despite these contributions, the technical complexity of implementing upgrade mechanisms and the risk of introducing new vulnerabilities remain significant barriers to broader adoption. Educational resources, user-friendly interfaces, and comprehensive best practice guidelines are essential to address these challenges. Accessible tutorials and transparency tools can empower developers to implement upgrade mechanisms effectively while informed users can better evaluate upgradeable contracts based on their risk tolerance and trust requirements.

Industry collaboration and standardization are equally important for promoting secure and reliable upgrades. Establishing shared standards, such as Ethereum Improvement Proposals (EIPs) specific to upgradeability, can encourage best practices and ensure platform interoperability. Engaging developers, auditors, and standards organizations in creating formal guidelines would further support consistent and secure upgrade practices within the blockchain landscape.

8.5 Open Questions and Future Directions

This research raises several open questions that highlight avenues for future exploration:

- **Governance of Smart Contract Upgrades:** How can governance frameworks balance transparency, security, and decentralization while providing developers the flexibility to address bugs or add functionalities? Research into decentralized autonomous organizations (DAOs) and their role in upgrade governance could provide valuable insights.

- **Integration of Predictive and Adaptive Technologies:** Can predictive models and adaptive mechanisms improve the security of upgradeable contracts? Utilizing the datasets and tools generated in this thesis, machine learning models could be developed for anomaly detection and predictive risk assessment. These proactive approaches would leverage historical upgrade patterns and vulnerability data to anticipate potential risks and recommend secure upgrade practices, particularly for high-risk applications.

- **Exploration of Emerging Upgrade Mechanisms:** While we focused on proxy-based approaches, emerging methods such as zero-knowledge proofs and cryptographic techniques could enhance the security and auditability of upgrades.

How can these methods be effectively applied to ensure that upgrades preserve or strengthen contract integrity?

- **Cross-Platform Compatibility and Interoperability:** As blockchain ecosystems diversify, compatibility and interoperability of upgrade mechanisms across platforms will be essential. What standards and protocols can facilitate secure upgrades in multi-chain environments?
- **User Trust and Perception:** How do users perceive upgradeable contracts, and what factors influence their trust? Investigating user attitudes toward upgradeability, transparency, and control could provide valuable insights for designing user-centered upgrade mechanisms that prioritize trust.

8.6 Final Remarks

This thesis has systematically addressed the complexities of smart contract upgradeability, balancing the need for adaptability with the principles of immutability that underpin blockchain technology. By establishing standardized definitions, documenting practical methods, and examining real-world adoption and security impacts, we have contributed to a deeper understanding of how smart contracts can be upgraded securely and transparently.

The tools and frameworks developed, PROXiFY and EvoChain, enhance transparency and security, empowering developers and users to navigate the challenges of upgrading smart contracts more effectively. Our findings reveal a cautious landscape where developers prioritize stability and security, often at the expense of flexibility, highlighting the need for ongoing support and innovation in upgrade practices.

The ability to upgrade smart contracts securely is essential for the sustained growth and acceptance of blockchain technologies, particularly as they become integral to diverse applications such as decentralized finance, supply chain management, and digital identity. By addressing the identified research gaps and achieving the outlined objectives, this thesis lays the groundwork for future advancements in smart contract upgradeability.

We envision a future where smart contracts can evolve to meet changing requirements and security challenges without compromising the trust and reliability that are the hallmarks of blockchain technology. Continued research and collaboration within the community will be crucial in realizing this vision, ensuring that blockchain systems remain robust, secure, and adaptable in the face of ongoing technological innovation.

Bibliography

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

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

[3] S. Sayeed, H. Marco-Gisbert, and T. Caira, "Smart contract: Attacks and protections", *IEEE Access*, vol. 8, pp. 24 416–24 427, 2020.

[4] B. K. Mohanta, S. S. Panda, and D. Jena, "An overview of smart contract and use cases in blockchain technology", in *2018 9th international conference on computing, communication and networking technologies (ICCCNT)*, IEEE, 2018, pp. 1–4.

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

[6] P. Qian, Z. Liu, Q. He, B. Huang, D. Tian, and X. Wang, "Smart contract vulnerability detection technique: A survey", *arXiv preprint arXiv:2209.05872*, 2022.

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

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

[9] M. M. Lehman, "Programs, life cycles, and the laws of software evolution", *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980. DOI: 10.1109/PROC.1980.11333.

[10] Z. Zheng *et al.*, "An overview on smart contracts: Challenges, advances and platforms", *Future Generation Computer Systems*, vol. 105, pp. 475–491, 2020. DOI: 10.1016/j.future.2019.12.019.

[11] M. I. Mehar, C. Shier, A. Giambattista, *et al.*, "Understanding a revolutionary and flawed grand experiment in blockchain: The dao attack", *Journal of Cases on Information Technology (JCIT)*, vol. 21, no. 1, pp. 19–32, 2019.

[12] D. Siegel, *The dao: A new era of crowdfunding*, https://hbr.org/2016/06/the-dao-a-new-era-of-crowdfunding, 2016.

[13] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts", in *International Conference on Security and Privacy in Communication Networks*, 2018. DOI: 10.1007/978-3-319-89722-6_10.

[14] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts", in *International Conference on Principles of Security and Trust (POST)*, 2017. DOI: 10.1007/978-3-662-54455-6_8.

[15] Y. Chen, Z. Zheng, E. C. H. Ngai, Q. Zheng, and Y. Zhou, "Soda: A generic online detection framework for smart contracts", in *Proceedings of the Network and Distributed System Security Symposium*, 2020. DOI: 10.14722/ndss.2020.24449.

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

[17] M. Salehi, J. Clark, and M. Mannan, "Not so immutable: Upgradeability of smart contracts on ethereum", in *International Conference on Financial Cryptography and Data Security*, Springer, 2022, pp. 539–554.

[18] V. C. Bui, S. Wen, J. Yu, X. Xia, M. S. Haghighi, and Y. Xiang, "Evaluating upgradable smart contract", in *2021 IEEE International Conference on Blockchain (Blockchain)*, IEEE, 2021, pp. 252–256.

[19] W. E. Bodell III, S. Meisami, and Y. Duan, "Proxy hunting: Understanding and characterizing proxy-based upgradeable smart contracts in blockchains", in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1829–1846.

[20] E. I. Proposals, *Eip-1967: Standard proxy storage slots*, Accessed: 2024-10-04, 2019. [Online]. Available: https://eips.ethereum.org/EIPS/eip-1967.

[21] E. I. Proposals, *Eip-1822: Universal upgradeable proxy standard (uups)*, Accessed: 2024-10-04, 2019. [Online]. Available: https://eips.ethereum.org/EIPS/eip-1822.

[22] Substack, *Answering some diamond questions*, Accessed: 2024-10-04, 2022. [Online]. Available: https://eip2535diamonds.substack.com/p/answering-some-diamond-questions.

[23] G. A. Oliva, A. E. Hassan, and Z. M. J. Jiang, "An exploratory study of smart contracts in the ethereum blockchain platform", *Empirical Software Engineering*, vol. 25, no. 3, pp. 1864–1904, 2020.

[24] C. Connors and D. Sarkar, "Review of most popular open-source platforms for developing blockchains", in *2022 fourth international conference on blockchain computing and applications (bcca)*, IEEE, 2022, pp. 20–26.

[25] M. Hamdaqa, L. A. P. Met, and I. Qasse, "Icontractml 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms", *Information and Software Technology*, vol. 144, p. 106 762, 2022.

[26] V. Braun and V. Clarke, "Using thematic analysis in psychology", *Qualitative Research in Psychology*, vol. 3, no. 2, pp. 77–101, 2006. DOI: 10.1191/1478088706qp063oa.

[27] J. Feist, N. Grech, and A. Groce, "Slither: A static analysis framework for smart contracts", in *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2019, pp. 8–15.

[28] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system", *Bitcoin.org*, 2008.

[29] V. Buterin, "A next-generation smart contract and decentralized application platform", *Ethereum White Paper*, 2014.

[30] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, "An overview of blockchain technology: Architecture, consensus, and future trends", in *2017 IEEE International Congress on Big Data (BigData Congress)*, IEEE, 2017, pp. 557–564.

[31] A. M. Antonopoulos and G. Wood, *Mastering Ethereum: Building Smart Contracts and DApps*. "O'Reilly Media, Inc.", 2018.

[32] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten, "SoK: Research perspectives and challenges for bitcoin and cryptocurrencies", in *2015 IEEE Symposium on Security and Privacy*, IEEE, 2015, pp. 104–121.

[33] M. Pilkington, "Blockchain technology: Principles and applications", *Research Handbook on Digital Transformations*, pp. 225–253, 2016.

[34] D. R. Stinson, *Cryptography: Theory and Practice*. CRC press, 2005.

[35] A. Gervais, G. O. Karame, K. Wust, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains", in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 3–16.

[36] V. Buterin, "Proof of stake: How i learned to love weak subjectivity", *Ethereum Blog*, 2014, `https://blog.ethereum.org/2014/11/25/proof-of-stake-a-decentralized-consensus-algorithm`.

[37] V. Buterin. "On public and private blockchains". (2015), [Online]. Available: `https//blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/`.

[38] V. Buterin and Visions, *Part 1: The Value of Blockchain Technology*. Ethereum Blog, 2015. [Online]. Available: `https//blog.ethereum.org/2015/04/13/visions-part-1-the-value-of-blockchain-technology/`.

[39] I. Allison, "Bank of england: Central banks looking at'hybrid systems' using bitcoin's blockchain technology", *International Business Time. July*, vol. 16, 2015.

[40] R. Brown, "Blockchain is where banks have the most obvious opportunity. but you ignore bitcoin at your peril", *Thought on the future of finance Blog*, pp. 1–1, 2015.

[41] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger", *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, 2014.

[42] E. Hildenbrandt, M. Saxena, N. Zhu, *et al.*, "Kevm: A complete formal semantics of the ethereum virtual machine", in *2018 IEEE Computer Security Foundations Symposium (CSF)*, IEEE, 2018, pp. 204–217.

[43] S. Team, *Solidity documentation*, `https://docs.soliditylang.org/`, Accessed:12-11-2024, 2020.

[44] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter", in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 254–269.

[45] C. Dannen, *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, 2017.

[46] T. Team, *Truffle suite*, `https://www.trufflesuite.com/`, Accessed: 12-11-2024, 2020.

[47] ZeppelinOS, *Upgrading smart contracts*, `https://docs.zeppelinos.org/docs/pattern.html`, Accessed: 12-11-2024, 2018.

[48] X. Xu, I. Weber, M. Staples, *et al.*, "A first look at the usability of ethereum smart contracts", in *IEEE Software*, vol. 35, 2018, pp. 83–88.

[49] N. Szabo, "The idea of smart contracts", *Nick Szabo's Papers and Concise Tutorials*, 1997.

[50] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things", *IEEE Access*, vol. 4, pp. 2292–2303, 2016.

[51] A. Miller, B. LaMacchia, and L. Kilmartin, "Smart contracts and their applications", *IEEE Security & Privacy*, vol. 17, no. 3, pp. 6–10, 2019.

[52] S. Tikhomirov, D. Voskresenskiy, I. Ivanitskiy, I. Tikhomirov, and R. Marchenko, "Smartcheck: Static analysis of ethereum smart contracts", in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[53] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)", in *International Conference on Principles of Security and Trust*, Springer, 2017, pp. 164–186.

[54] B. Jiang, Y. Liu, W.-K. Chan, and T. Su, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection", in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 259–269.

[55] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale", in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 653–663.

[56] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts", in *Proceedings of the ACM on Programming Languages*, vol. 2, 2018, pp. 1–27.

[57] C. Torres, M. Steichen, R. Norvill, and R. State, "The art of the scam: Demystifying honeypots in ethereum smart contracts", in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1591–1607.

[58] C. Torres, M. Steichen, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts", in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 664–676.

[59] G. Grieco, I. Nikolić, A. Kolluri, *et al.*, "Echidna: Effective, usable, and fast fuzzing for smart contracts", in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 545–560.

[60] S. Park, J. Heo, and M. Choi, "Formal verification of blockchain smart contracts using Coq", in *Proceedings of the 1st ACM Workshop on Formal Methods in Security*, 2018, pp. 31–36.

[61] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab", in *International Conference on Financial Cryptography and Data Security*, Springer, 2016, pp. 79–94.

[62] X. Zhang, X. Li, Y. Xue, and L. Liu, "Smart contract security: A software lifecycle perspective", *IEEE Security & Privacy*, vol. 18, no. 3, pp. 92–96, 2020.

[63] O. Team, *Openzeppelin contracts*, `https://docs.openzeppelin.com/contracts/`, 2020.

[64] M. M. Lehman, "Laws of software evolution revisited", *Lecture Notes in Computer Science*, vol. 1149, pp. 108–124, 1996.

[65] M. M. Lehman and J. F. Ramil, "Metrics and laws of software evolution—the nineties view", *Proceedings of the Fourth International Software Metrics Symposium*, pp. 20–32, 1997.

[66] T. Mens and S. Demeyer, *Software Evolution*. Springer, 2008.

[67] J. Chen, X. Xia, D. Lo, J. Grundy, and X. Yang, "Maintenance-related concerns for post-deployed ethereum smart contract development: Issues, techniques, and future challenges", *Empirical Software Engineering*, vol. 26, no. 6, pp. 1–44, 2021.

[68] B. P. Lientz and E. B. Swanson, *Software maintenance management*. Addison-Wesley Longman Publishing Co., Inc., 1980.

[69] M. W. Godfrey and D. M. German, "The past, present, and future of software evolution", in *2008 Frontiers of Software Maintenance*, IEEE, 2008, pp. 129–138.

[70] I. Weber, X. Xu, R. Riveret, G. Governatori, A. Ponomarev, and J. Mendling, "Untrusted business process monitoring and execution using blockchain", in *International Conference on Business Process Management*, Springer, 2016, pp. 329–347.