Micro-Patterns in Solidity Code

Luca Ruschioni
luca.ruschioni@unicam.it
University of Camerino
Camerino, Italy

Robert Shuttleworth
robert.shuttleworth2@brunel.ac.uk
Brunel University of London
London, UK

Rumyana Neykova
rumyana.neykova@brunel.ac.uk
Brunel University of London
London, UK

Barbara Re
barbara.re@unicam.it
University of Camerino
Camerino, Italy

Giuseppe Destefanis
giuseppe.destefanis@brunel.ac.uk
Brunel University of London
London, UK

Abstract

Solidity is the predominant programming language for blockchain-based smart contracts, and its characteristics pose significant challenges for code analysis and maintenance. Traditional software analysis approaches, while effective for conventional programming languages, often fail to address Solidity-specific features such as gas optimization and security constraints. This paper introduces micro-patterns - recurring, small-scale design structures that capture key behavioral and structural peculiarities specific to a language - for Solidity language and demonstrates their value in understanding smart contract development practices. We identified 18 distinct micro-patterns organized in five categories (Security, Functional, Optimization, Interaction, and Feedback), detailing their characteristics to enable automated detection. To validate this proposal, we analyzed a dataset of 23258 smart contracts from five popular blockchains (Ethereum, Polygon, Arbitrum, Fantom and Optimism). Our analysis reveals widespread adoption of micro-patterns, with 99% of contracts implementing at least one pattern and an average of 2.76 patterns per contract. The *Storage Saver* pattern showed the highest adoption (84.62% mean coverage), while security patterns demonstrated platform-specific adoption rates. Statistical analysis revealed significant platform-specific differences in adoption, particularly in *Borrower*, *Implementer*, and *Storage Saver* patterns.

CCS Concepts

• **Software and its engineering → Automated static analysis**.

Keywords

Solidity, Smart Contract, Micro Pattern, Pattern Recognition

ACM Reference Format:

Luca Ruschioni, Robert Shuttleworth, Rumyana Neykova, Barbara Re, and Giuseppe Destefanis. 2025. Micro-Patterns in Solidity Code. In *Proceedings of The 29th International Conference on Evaluation and Assessment in Software Engineering (EASE 2025).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

EASE 2025, 978-1-4503-XXXX-X/18/06

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Solidity is a statically-typed, object-oriented programming language designed for implementing smart contracts on blockchain platforms. Since its release in 2015, Solidity has become the primary language for blockchain development [1], supporting platforms such as Ethereum, Polygon, Arbitrum, Fantom, and Optimism, which are object of our study. Its distinct features [7], including gas optimization [17], strict state management, and security constraints, introduce significant challenges for code analysis and quality assessment [33]. Traditional software metrics [13], though effective for general-purpose programming languages, are often inadequate for evaluating Solidity's unique properties, such as gas consumption, state variable organization, and access control mechanisms [8, 10].

Micro-patterns, defined as small and recognizable programming constructs that reflect intentional design decisions, were first introduced by Gil and Maman for Java [15]. In their study, Gil and Maman demonstrated that 75% of the classes in a Java system match at least one micro-pattern from their defined catalog. Unlike conventional software metrics, micro-patterns offer structured insights into code quality and evolution by capturing common structural and behavioral elements. Building on this concept, we propose a novel framework for identifying and analyzing micro-patterns in Solidity code. We introduce a catalog of Solidity-specific micro-patterns that encapsulate its distinct programming characteristics. This work is the first systematic effort to apply micro-patterns to Solidity, providing a structured approach to evaluating smart contract. In detail, we address the following research questions:

(RQ1) What micro-patterns can be identified in Solidity code, and how can they be systematically detected? We define a catalog of recurring micro-patterns in Solidity language and develop a framework for their automated detection.

(RQ2) To what extent are micro-patterns adopted in smart contract development? We analyze the prevalence of micro-patterns in a large dataset of smart contracts to understand their role in development practices.

(RQ3) How do micro-pattern frequencies vary across different blockchain platforms? We examine how micro-pattern adoption differs across platforms, highlighting variations and commonalities in development strategies.

(RQ4) Which relationships exist between different micro-patterns, and how are they spread? Through correlation and exclusivity analysis, we investigate how micro-patterns co-occur and interact in Solidity development.

Our contributions are as follows. First, we introduce a catalog of 18 micro-patterns grouped into five categories: *security, functional, optimization, interaction, and feedback*. Each micro-pattern is formally defined to enable automated detection and to reflect critical aspects of Solidity development. Second, we develop a micro-pattern detection framework with 93% success rate across different Solidity versions. Our framework enables large-scale analysis of contracts on any Solidity-compatible blockchain and generates comprehensive metrics including pattern frequency, coverage rates, and cross-pattern correlation measures. Third, we conduct an empirical study of over 23258 smart contracts from the five most popular blockchains. Our frequency analysis shows that 99% of contracts exhibit at least one micro-pattern, with an average of 2.76 micro-patterns per contract. The correlation analysis reveals that micro-patterns represent independent design choices, with predominantly weak relationships ($\phi < 0.15$) between pattern pairs. We also perform a cross-platform comparison identifying significant variations in pattern adoption, particularly in optimization and security, reflecting platform-specific development practices and constraints. All materials presented in this paper, including the micro-pattern detection framework, datasets, and analysis scripts, are fully replicable and verifiable in our replication package **link**.

The remainder of this paper is organized as follows. Section 2 reviews related work on micro-patterns and smart contract analysis. Section 3 introduces our catalog of 18 micro-patterns and identification method, while Section 4 describes our framework for automated detection. Section 5 covers data collection and preparation, followed by Section 6 with the empirical evaluation. Section 7 discusses threats to validity and Section 8 concludes the paper.

2 Related Work

Research on micro-patterns has evolved from their initial definition for Java to applications in code quality analysis. We survey these developments alongside existing approaches to smart contract analysis to contextualize the current understanding of code structures in both traditional and blockchain environments.

Evolution of Micro-Pattern Analysis. Micro-patterns are advanced through several key developments in detection and classification methods. Arcelli et al. [19] redefined micro-patterns in terms of number of methods (NOM) and number of attributes (NOA) of a class. The relationship between micro-patterns and programming practices [6], and code quality emerged as another important research direction. Kim et al. [16] examined micro-pattern evolution across software versions, identifying specific evolution types associated with increased defect rates. Building on this work, Destefanis et al. [9] analyzed multiple Eclipse releases and found that classes not matching any micro-pattern showed higher fault rates than those exhibiting patterns. These findings about pattern presence shaped our method for evaluating Solidity code quality.

Micro-Patterns for Quality and Security Assessment. The application of micro-patterns has extended into security analysis, with researchers demonstrating their value for vulnerability detection. Sultana et al. [26, 27] identified correlations between certain patterns and security vulnerabilities in Apache Tomcat, showing how pattern analysis could enhance traditional vulnerability prediction methods that relied solely on software metrics. This security-focused application aligns with our work, as smart contracts face security challenges that can benefit from pattern-based analysis. Codabux et al. [4] further demonstrated the value of combining pattern analysis with other techniques by investigating relationships between traceable patterns and code smells. Their work indicated that certain patterns may signal design issues, a finding that influenced our identification of problematic patterns in smart contracts. While these studies show the utility of micro-patterns in code analysis, their findings stem from object-oriented programming concepts specific to traditional software environments.

Smart Contract Analysis Techniques. Current smart contract analysis research has focused primarily on static analysis of code structure and dynamic analysis of runtime properties. Ghaleb et al. [14] evaluated six static analysis tools through systematic bug injection, revealing limitations in detecting known vulnerability patterns. Their work demonstrates gaps in current tools, but focuses on vulnerability detection rather than identifying intentional design decisions in smart contract development practices. Static analysis tools employ various approaches. SmartCheck uses pattern matching through XPath expressions [28], while Slither [12] exploits AST analysis to discover a predefined set of vulnerabilities. Instead, other solutions such as Mythril [11] and Manticore apply symbolic execution [21] to reproduce smart contract execution and found possible issues. As highlighted by Vidal et al. [31], while these tools can detect predefined vulnerabilities, they struggle with more complex scenarios that involve multiple contracts.

Research has extensively explored formal verification for smart contracts, with formal methods being particularly effective despite limited adoption [22, 24, 29]. The focus has been primarily on functionality verification, with fewer works addressing security specifically. While these works provide correctness guarantees, they do not address the identification of common programming patterns that could guide better development practices. As noted by Atzei et al. [2], there remains a gap between theoretical security properties and practical development patterns. Traditional software metrics have been adapted for smart contract evaluation, but often fail to capture blockchain concerns like gas optimization and state management [30]. Recent studies identify authorization and authentication as major security risks in Ethereum smart contracts, especially those with external dependencies [3]. Destefanis et al.'s work [9] on detecting problematic patterns through metrics suggests approaches that could be modified for smart contracts.

Research Gap and Our Contributions. The existing literature demonstrates both the value of micro-pattern analysis in traditional software and the need for specialized evaluation techniques for smart contracts. Current approaches apply patterns that do not translate well to blockchain environments, or analyze smart contracts without systematic pattern recognition. Some studies examine contract design aspects in isolation, missing the broader context of how patterns interact. We build upon Gil and Maman's [15] approach to micro-pattern definition while addressing the distinct characteristics of Solidity programming. Unlike previous adaptations of object-oriented metrics, our approach targets Solidity-specific concerns such as gas costs optimization and security.

Table 1: Solidity Micro-Patterns Catalog

Category	Name	Description	Entity Types
Security	**Ownable**	An entity that maintains an owner's address and restricts access to specific functions using modifiers.	Contract
	Stoppable	An entity with a toggable state, so it can be paused, resumed, or permanently stopped if necessary.	Contract
	Pull Payment	A payment model where recipients withdraw funds themselves instead of receiving from the contract.	Contract
	Reentrancy Guard	A modifier that prevents reentrancy attacks by ensuring state updates occur before any external calls.	Contract
Functional	**Payable**	An entity that includes `fallback()` and `receive()` functions to enable the receipt of funds.	Contract
	Borrower	An entity that utilizes external libraries to perform predefined operations on specific data types.	Contract
	Implementer	An entity that implements all functions inherited from interfaces or abstract contracts.	Contract
	Modifier Usage	An entity that leverages reusable modifiers to enforce common conditions across multiple functions.	Contract, Library
Optimization	**Storage Saver**	An entity that minimizes storage costs by efficiently arrange state variables into fewest storage slots.	Contract
	Reader	An entity where all functions are defined as `view`, ensuring they do not modify the contract state.	Contract, Interface, Library
	Operator	An entity where all functions are defined as `pure`, so they do not read or modify any state variable.	Contract, Interface, Library
Interaction	**Provider**	An entity where all functions are `external` so that only external deployed contracts can invoke them.	Contract, Interface, Library
	Supporter	An entity where all functions are `internal` so that only the contract and its inheritors can use them.	Contract, Interface, Library
	Delegator	An entity that delegates operations by invoking functions in another deployed contract.	Contract, Library
Feedback	**Named Return**	An entity where return values are explicitly assigned names in the function definition.	Contract, Interface, Library
	Returnless	An entity that does not return any values from its functions.	Contract, Interface, Library
	Emitter	An entity where every function execution emits at least one event.	Contract, Library
	Muted	An entity where no events are emitted during the execution of its functions.	Contract, Library

3 Micro-Pattern Analysis

This section presents our method for identifying and categorizing micro-patterns in Solidity, addressing RQ1's focus on recognizable development micro-patterns. We begin by describing the identification process, followed by an organized catalog based on functional roles in Solidity development. Finally, we introduce our framework for automated detection.

3.1 Micro-Pattern Identification Process

Our method for identifying micro-patterns in Solidity builds upon Gil and Maman's [15] systematic process while adapting it for blockchain-specific development practices[1]. First, we analyzed how Gil and Maman's Java micro-patterns could be translated to Solidity's domain. We examined each micro-pattern in their catalog and evaluated whether it could meaningfully capture Solidity development practices. This revealed that while some micro-patterns (like *Implementor*) could transfer directly, blockchain's characteristics required new patterns.

Second, we conducted an iterative pattern discovery process:

(1) We examined Solidity's distinctive features (e.g., modifiers, view/pure functions, event emissions) and considered meaningful restrictions on their usage;
(2) For each potential micro-pattern, we analyzed its impact in addressing specific concerns like "what are the implications of state variable arrangements on gas costs?";
(3) We implemented initial micro-pattern definitions for applying them to public contract repositories and evaluate their behaviours;
(4) Through manual code inspection of matched contracts, we refined definitions, merged similar micro-patterns, and discarded those that did not capture design decisions;
(5) When automatic pattern detection revealed clusters of similar but not identical implementations, we analyzed whether these represented meaningful variants requiring pattern refinement.

[1]https://docs.soliditylang.org/en/latest/style-guide.html

Finally, we validated all candidate micro-patterns against three key criteria defined in the following:

Design Intent. A micro-pattern must capture a meaningful architectural or functional aspect in Solidity development design practices. For instance, the *Storage Saver* micro-pattern (Table 1) reflects an optimization strategy where developers arrange state variables to minimize storage slots, reducing gas costs. This distinguishes meaningful micro-patterns from arbitrary structures. For example, a contract that increments a counter or uses simple conditionals may create recognizable patterns, but these do not represent deliberate design decisions.

Mechanical Recognizability. A micro-pattern must be expressible as a condition that can be verified through static analysis [15]. For example, verifying whether all state-modifying functions implement a reentrancy guard can be achieved by analyzing the presence of specific modifiers and function calls in the code. We formalized each micro-pattern using first-order logic (FOL) specifications, defining precise conditions on contract components, methods, and modifiers (see Appendix A: Table 4). This formal definition ensures automatic and unambiguous detection.

Empirical Validation. A micro-pattern must be observable in deployed smart contracts across multiple blockchain platforms. We validated each micro-pattern's adoption through analysis of contracts, ensuring that it represents a recurring development practice rather than an isolated case. This empirical validation distinguishes meaningful micro-patterns implemented with a specific scope from coincidental code structures.

Following these guidelines, our method balances automated detection with the need to capture meaningful design choices. The combination of mechanical recognizability, design intent, and empirical evidence ensures that the identified micro-patterns represent genuine development practices addressing challenges in Solidity smart contracts. Patterns that were infrequent, highly contract-specific, or not mechanically detectable were excluded.

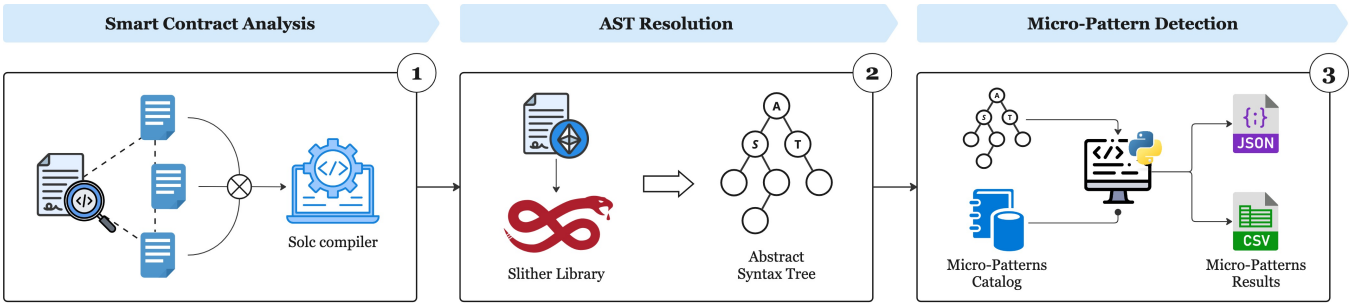

Figure 1: Micro-Pattern Detection Framework

3.2 Micro-Patterns Catalog

After following the identification process, we defined 18 micro-patterns reported in Table 1. While additional ones may exist, this catalog contains the most frequent and practically relevant ones based on our criteria. The categorization reflects primary concerns in smart contract development, including *security*, *gas costs optimization*, and *cross-contract interactions*.

Security Patterns. The immutable nature of deployed smart contracts requires built-in security measures. Following Gil and Maman's observation that micro-patterns can capture language-specific features, these structures reflect different approaches to address unique security requirements of smart contracts compared to traditional software. For example, the *Ownable* micro-pattern implements single-owner authorization, restricting functions execution. Instead, *Pull Payment* and *Reentrancy Guard* implement protection mechanisms that, once deployed, cannot be modified.

Functional Patterns. Similar to how Java micro-patterns capture core operational features, this category encompasses key smart contract behaviors. These include micro-patterns such as *Pull Payment* that enables smart contracts to receive funds and modularity mechanisms such as *Borrower* and *Implementer*. They demonstrate how to leverage Solidity features to create maintainable and extensible contracts.

Optimization Patterns. The gas-cost model of the Ethereum Virtual Machine creates unique optimization requirements. These micro-patterns focus on improving resource efficiency through techniques like the *Storage Saver* for minimizing storage costs. This category has no direct parallel in traditional micro-patterns, reflecting blockchain-specific concerns.

Interaction Patterns. Smart contracts rarely operate in isolation. Interaction micro-patterns govern contract communication and functionality delegation. Unlike object-oriented micro-patterns that focus on class relationships, these structures define standard approaches for inter-contract communication in the decentralized blockchain ecosystem.

Feedback Patterns. The final category addresses how smart contracts communicate state changes and execution results. Through event emission and return values, these micro-patterns define how smart contracts interact with external systems. This category reflects the need for observable behavior in decentralized applications.

The organization into these categories emerges from both the intrinsic properties of the Solidity language and the practical requirements of smart contract development.

(RQ1) What micro-patterns can be identified in Solidity code, and how can they be systematically detected? Through analysis of Solidity features and development practices, we identified 18 micro-patterns across five categories. Each micro-pattern represents a recognizable and reproducible development practice. The catalog provides the foundation for understanding and implementing common smart contract functionalities, with micro-patterns applicable across different blockchain platforms and contract types.

4 Micro-Pattern Detection Framework

We implemented a Python-based framework, whose workflow is depicted in Figure 1, to analyze Solidity files and detect micro-patterns in the code. It supports both single-file contracts and those organized across multiple sources, manages compiler version resolution for different Solidity compiler versions, and handles dependency management by resolving import paths and ensuring the availability of required dependencies.

The detection process begins by scanning the input paths to identify all Solidity files and their associated dependencies, as shown in Figure 1 **(step 1)**. Once the files are identified, the tool analyzes them to extract the latest compatible Solidity compiler versions required for each contract, resolving the versions based on pragma directives. To optimize the compilation process, the tool orders the files by compiler version, reducing unnecessary compiler switches and improving efficiency.

After determining the compilation order, the process continues as presented in Figure 1 **(step 2)**, where the framework uses the Slither library[2] [12] to compile each smart contract and extract the Abstract Syntax Tree (AST). The framework implements our formal First-order-logic (FOL) specifications through AST traversal operations. Each pattern detection algorithm systematically checks AST nodes according to the conditions defined in our FOL specifications. The implementation's correctness is validated through testing against known pattern examples.

With the AST representation of the code, the framework traverses the structure to identify occurrences of predefined micro-patterns in each entity, which may include contracts, abstract contracts, interfaces, and libraries. The detection process not only analyzes individual entities but also considers inheritance hierarchies,

[2]Slither library: https://github.com/crytic/slither/

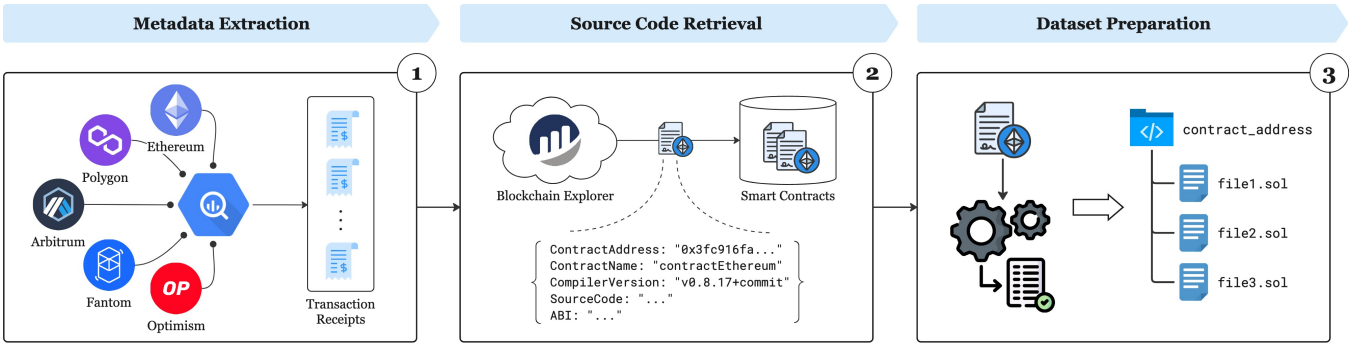

Figure 2: Data Collection Method

modifiers, and external dependencies to ensure accurate results. Dependencies such as imported contracts and external libraries are automatically resolved and analyzed alongside the provided contracts, ensuring that all necessary components are available for micro-pattern detection.

For each analyzed entity, the tool collects and outputs detailed information as in Figure 1 **(step 3)**, including the contract's name, file path, compiler version, type (i.e., contract, abstract contract, interface, or library), and whether each micro-pattern is present. The results are then exported in the user-specified format, enabling further analysis of micro-pattern prevalence and distribution across the analyzed smart contracts.

Our framework achieved a 93% success rate in processing 23258 out of 25000 verified smart contracts. The unsuccessful cases were primarily due to dependency resolution issues and import path remapping problems in the downloaded smart contract sources, rather than limitations in the pattern detection logic itself. Overall, this framework ensures accurate and efficient detection of micro-patterns while adapting to the diverse structures and requirements of Solidity projects.

5 Data Collection

To empirically validate the micro-pattern catalog and address our research questions, we follow the method in Figure 2 to collect a large dataset of verified smart contracts, including both metadata and source code. Verified smart contracts are those whose source code is available and validated in blockchain explorers[3] - web platforms that index and display blockchain transaction data. Our detection framework, described in Section 4, requires this source code to accurately identify micro-pattern occurrences through static analysis. This dataset supports our empirical evaluation of micro-pattern adoption (RQ2), distribution across different blockchain platforms (RQ3), and relationships between micro-patterns (RQ4).

5.1 Metadata Extraction

We extracted all transaction receipts, from the genesis block up to 31st December 2024, from five blockchain networks: Ethereum, Polygon, Arbitrum, Fantom, and Optimism. These networks were selected based on their representation of deployed contracts and availability in Google BigQuery Web3 public datasets[4].

As in Figure 2 (step 1), we retrieved contract metadata from transaction receipts with non-null `contract_address` fields on BigQuery database, indicating successful deployment of a smart contract. For each smart contract, we collect the following information: contract address, creator address, block timestamp, block number, and transaction hash. Contract and creator addresses allow tracking contract origins and ownership micro-patterns. Block number and timestamp, instead, provide temporal context for micro-pattern evolution analysis. Finally, transaction hash serves as a unique identifier and enable verification of deployment details.

Table 2: Blockchain Entities Statistics

Blockchain	Smart Contracts	Entities				
		Contract	Abstract C.	Interface	Library	Total
Ethereum	4.855	10.097	10.159	20.092	10.712	51.060
Polygon	4.900	11.666	16.664	23.341	12.092	63.763
Arbitrum	4.415	34.935	24.113	46.382	25.328	130.758
Fantom	4.871	11.154	12.437	30.657	14.256	68.504
Optimism	4.217	7.708	10.039	18.732	11.243	47.722
Total	23.258	75.560	73.412	139.204	73.631	361.807

5.2 Source Code Retrieval

From the extracted metadata, we collected an adequate number of verified smart contracts per platform, obtaining their source code through blockchain explorers (Etherscan, Polygonscan, Arbiscan, Ftmscan, and Optimistic)[5] as shown in Figure 2 (step 2). The sample size must provide statistical power for analyzing micro-pattern prevalence (RQ2), cross-blockchain comparisons (RQ3), and correlation analysis (RQ4).

To determine the minimum required sample size of our dataset, we conducted a statistical power analysis for chi-square tests used in comparing micro-pattern distributions across platforms. With five blockchains, our analysis involves ten pairwise comparisons ($\frac{5(5-1)}{2} = 10$), necessitating a Bonferroni-corrected significance level of $\alpha = 0.005$ (0.05/10) [32]. Power analysis indicated that detecting small effect sizes ($w = 0.1$) requires a minimum of 1332 smart

[3]Verified smart contracts list: https://etherscan.io/contractsVerified

[4]Google Web3 Datasets: https://cloud.google.com/application/web3/discover/

[5]Explorer URLs: https://etherscan.io/, https://polygonscan.com/, https://arbiscan.io/, https://ftmscan.com/, https://optimistic.etherscan.io/

$$E = \{\, e \mid \text{entity } e \in \text{Dataset} \wedge \text{type}(e) \in \{\texttt{contract}, \texttt{library}, \texttt{interface}\}\}$$

$$MP = \{\, mp \mid mp \text{ is one of the defined micro-patterns}\}$$

$$\text{ValidTypes}(mp) \subseteq \{\texttt{contract}, \texttt{interface}, \texttt{library}\}$$

$$M(mp, e) = \begin{cases} 1, & \text{if type}(e) \in \text{ValidTypes}(mp) \text{ and } e \text{ satisfies } mp, \\ 0, & \text{otherwise.} \end{cases}$$

$$\text{Frequency}(mp) = \sum_{e \in E(mp)} M(mp, e)$$

$$\text{Coverage}(mp) = \frac{\text{Frequency}(mp)}{|E(mp)|}$$

$$\text{Prevalence}(mp) = \frac{\text{Frequency}(mp)}{\sum_{mp' \in MP} \text{Frequency}(mp')}$$

Figure 3: Analysis framework. Left: *E* captures entities (contracts, interfaces, libraries) from these projects, *MP* defines micro-patterns, *ValidTypes* maps patterns to applicable entity types, and *M* is the matching function. Right: Core metrics measure pattern occurrence (*Frequency*), adoption rate among eligible entities (*Coverage*), and relative dominance (*Prevalence*).

contracts per blockchain. We chose to collect at least 4000 smart contracts for each blockchain to enhance the statistical validity of our analysis in several ways. First, this larger sample size enables detection of effects smaller than $w = 0.1$, allowing identification of subtle variations in micro-pattern adoption across platforms. Second, when examining relationships among 18 micro-patterns (RQ4), the increased sample size reduces variance in correlation estimates, particularly for weak associations. Finally, given the heterogeneous nature of smart contract deployment across blockchain platforms, having at least 4000 smart contracts per platform provides more reliable coverage of diverse implementation patterns.

As detailed in Table 2, our dataset encompasses both standalone smart contracts and decentralized applications (DApps). Standalone smart contracts implement specific functionalities such as token management or voting systems, while DApps comprise multiple interconnected smart contracts with complex state management requirements. This architectural diversity strengthens our analysis of micro-pattern relationships (RQ4) and design decisions (RQ2). Moreover, the dataset includes smart contracts implemented with different Solidity versions across the five blockchain platforms, ranging from compiler version 0.4.x to 0.8.x. We organized these in blockchain-specific SQL databases to facilitate efficient micro-pattern detection and cross-chain analysis for RQ3, enabling both granular and comparative studies of micro-pattern adoption.

5.3 Dataset Preparation

After retrieving the smart contracts' source code, we processed and organized the files to preserve their dependency relationships, as illustrated in Figure 2 (step 3). This preparation phase addressed three key challenges in Solidity compilation: version compatibility, dependency resolution, and multi-file project organization. We processed each smart contract differently based on its source code structure. For single-file smart contracts, we maintain their original structure while verifying version compatibility. On the other hand, we implement a systematic dependency resolution mechanism for multi-file smart contracts that reconstructs the project hierarchy. This process analyzes Solidity import statements and remaps paths to maintain the smart contract's intended structure, handling both direct dependencies and custom library paths.

Based on this, we created a structured dataset for each of the selected blockchain. We stored the processed smart contracts in separate directories, each containing a complete and self-contained project. The directory structure that preserves the relationships between smart contract components while resolving potential conflicts in import paths and library references. This organization ensures that when the Abstract Syntax Tree (AST) is generated for micro-pattern detection (Section 4), all inheritance hierarchies and external dependencies are properly resolved for the analysis. The complete dataset, including all processed smart contracts and their source code, is available in our replication package **link**.

6 Empirical Evaluation

This section presents the empirical evaluation of micro-patterns in Solidity language, analyzing pattern adoption rates (RQ2), cross-platform variations (RQ3), and pattern relationships (RQ4) across smart contracts in our dataset. This evaluation demonstrates the prevalence of identified micro-patterns in real-world smart contract development, quantifies relationships between different micro-patterns, and reveals how developers adopt them across different blockchain platforms. Through systematic analysis of our dataset, we examine both the individual characteristics of each micro-pattern and their collective impact on smart contract development.

6.1 Pattern Adoption Analysis (RQ2)

In the first part of the evaluation, we examined the extent of micro-pattern adoption in smart contract development. We analyzed statistical metrics to understand the frequency and distribution of micro-patterns in Solidity smart contracts. Figure 3 formalizes our analysis framework: the set E captures all code entities across our dataset, where each entity (of type `contract`, `interface`, or `library`) may implement one or more micro-patterns, with pattern eligibility defined by *ValidTypes*; the matching function M returns a binary value indicating whether an eligible entity implements a specific micro-pattern (1) or not (0).

We evaluated micro-pattern adoption using three complementary metrics. *Frequency* counts the absolute number of entities matching a pattern. *Coverage* normalizes this count by the number

Table 3: Metrics for Micro-Patterns across Blockchains with Coverage Statistics. For each blockchain, blue-shaded cells indicate micro-patterns with the highest adoption rates, while azure-shaded cells micro-patterns with the lowest adoption.

Category	Micro-Pattern	Ethereum			Polygon			Arbitrum			Fantom			Optimism			Total Cov. Stats (%)	
		Freq.	Cov. (%)	Prev. (%)	Freq.	Cov. (%)	Prev. (%)	Freq.	Cov. (%)	Prev. (%)	Freq.	Cov. (%)	Prev. (%)	Freq.	Cov. (%)	Prev. (%)	Mean±σ	Median
Security	Ownable	269	1.33	0.20	743	2.62	0.41	446	0.76	0.11	947	4.01	0.53	98	0.55	0.07	1.85±1.30	1.33
	Stoppable	448	2.21	0.33	1328	2.25	0.34	1328	2.25	0.26	368	1.56	0.21	436	2.46	0.33	2.15±0.31	2.25
	Pull Payment	61	0.30	0.04	32	0.11	0.02	47	0.08	0.01	21	0.09	0.01	32	0.18	0.02	0.15±0.08	0.11
	Reentrancy Guard	39	0.19	0.03	29	0.10	0.02	119	0.20	0.03	101	0.43	0.06	8	0.05	0.01	0.19±0.13	0.19
Functional	Payable	1040	5.13	0.76	2452	8.66	1.37	2153	3.65	0.55	3112	13.19	1.75	2448	13.79	1.86	8.88±4.10	8.66
	Borrower	7961	39.30	5.80	11818	41.72	6.60	37018	62.69	9.43	8225	34.86	4.62	8418	47.43	6.40	45.20±9.64	41.72
	Implementer	7942	39.21	5.79	12233	43.18	6.83	35219	59.64	8.97	10363	43.93	5.83	8111	45.70	6.17	46.33±6.98	43.93
	Modifier Usage	10417	33.64	7.60	14211	35.16	7.94	31114	36.88	7.93	10194	26.93	5.73	7879	27.18	5.99	31.96±4.13	33.64
Optimization	Storage Saver	16301	80.47	11.89	25969	91.67	14.50	42195	71.46	10.75	21854	92.64	12.29	15413	86.85	11.72	84.62±7.87	86.85
	Reader	6857	13.43	5.00	9956	15.61	5.56	13473	10.30	3.43	8782	12.82	4.94	6391	13.39	4.86	13.11±1.70	13.39
	Operator	4743	9.29	3.46	4551	7.14	2.54	9184	7.02	2.34	4066	5.94	2.29	3945	8.27	3.00	7.53±1.15	7.14
Interaction	Provider	18422	36.08	13.43	20386	31.97	11.38	44519	34.05	11.34	26128	38.14	14.69	16830	35.27	12.80	35.10±2.06	35.27
	Supporter	11123	21.78	8.11	11480	18.00	6.41	22751	17.40	5.80	12606	18.40	7.09	8843	18.53	6.72	18.82±1.53	18.40
	Delegator	13188	42.59	9.62	18789	46.48	10.49	49541	58.71	12.62	17794	47.02	10.01	14163	48.85	10.77	48.73±5.39	47.02
Feedback	Named Return	16225	31.78	11.83	17754	27.84	9.91	46541	35.59	11.86	23896	34.88	13.44	15780	33.07	12.00	32.63±2.75	33.07
	Returnless	4850	9.50	3.54	7357	11.54	4.11	12611	9.64	3.21	7574	11.06	4.26	5598	11.73	4.26	10.69±0.94	11.06
	Emitter	191	0.62	0.14	121	0.30	0.07	287	0.34	0.07	150	0.40	0.08	139	0.48	0.11	0.43±0.11	0.40
	Muted	17075	55.14	12.45	20486	50.68	11.44	44036	52.19	11.22	21661	57.23	12.18	16965	58.52	12.90	54.75±2.95	55.14
Total Coverage (%)			99.98			99.99			99.99			100.00			99.98		99.99±0.01	

of eligible entities, revealing the percentage of potential implementations that adopt the pattern. *Prevalence* contextualizes each pattern's adoption by showing its frequency relative to all pattern occurrences, indicating its relative importance in the ecosystem.

Table 3 presents the adoption metrics across five major blockchain platforms. Our analysis revealed widespread adoption of micro-patterns in smart contract development, with projects implementing an average of 2.76 patterns (median: 2.00, σ: 1.43). The consistent adoption across platforms (means ranging from 2.60 to 3) suggests these patterns represent fundamental development practices. *The Storage Saver* pattern showed the highest coverage at 84.62% (σ: 7.87%). Its prevalence rates (10.75-14.50%) align with what we would expect given the average number of patterns per contract, suggesting it is a fundamental pattern that is consistently implemented alongside other patterns rather than being used in isolation.

Approximately 28% (5 out of 18) of our identified micro-patterns achieved wide adoption (>40% coverage), distributed across functional (*Borrower*: 45.20%, *Implementer*: 46.33%), optimization (*Storage Saver*: 84.62%), interaction (*Delegator*: 48.73%), and feedback (*Muted*: 54.75%) categories. Security-focused patterns showed lower but consistent adoption rates, with *Reentrancy Guard* (mean: 0.19%, σ: 0.13%) and *Pull Payment* (mean: 0.15%, σ: 0.08%) being selectively implemented. These stable rates across platforms indicated deliberate pattern selection for specific security requirements rather than oversight. The varying adoption rates, from nearly universal optimization patterns to selective security patterns, demonstrated that our catalog successfully captured both foundational and specialized development practices in the Solidity ecosystem.

While our method considers pattern frequency as one indicator of meaningful development practices, we acknowledge that high adoption rates do not necessarily indicate best practices. For instance, the *Muted* pattern's high coverage (54.75%) might reflect insufficient logging practices rather than optimal design choices. Similarly, some infrequent patterns might represent important security practices that are underutilized. Therefore, our identification process focuses on capturing recurring structural choices in smart contract development, whether beneficial or potentially problematic, to provide a foundation for future research on pattern impact and quality outcomes.

> ***(RQ2) To what extent are micro-patterns adopted in smart contract development?*** Our identified micro-patterns are widely adopted, with projects implementing an average of 2.76 patterns (median: 2.00, σ: 1.43). We found that 28% of them achieve high adoption rates (>40% coverage), distributed across *Functional, Optimization, Interaction*, and *Feedback* categories, while others serve more specialized purposes. This distribution suggests these micro-patterns effectively capture both foundational and specialized smart contract development practices.

6.2 Pattern Distribution Analysis (RQ3)

To understand how blockchain environments influence design approaches, we analyzed micro-pattern distribution across platforms using both descriptive statistics and inferential analysis. With 4000+ smart contracts per blockchain platform (see Table 2), we examined pattern adoption variations across platforms and then employed non-parametric statistical tests to validate observed differences. We used non-parametric tests [23] because our data consisted of binary observations (presence/absence of micro-patterns) and not follow normal distribution assumptions required for parametric tests.

The descriptive analysis in Table 3 revealed distinct platform-specific adoption trends. Security micro-patterns showed notably different adoption rates, with Ethereum exhibiting higher coverage of security-focused patterns. For instance, *Reentrancy Guard* showed higher coverage on Ethereum (0.19%) compared to Optimism (0.05%), while *Pull Payment* patterns had higher coverage on Ethereum (0.30%) than other platforms (mean: 0.11%). This aligned with Ethereum's role as the primary platform for high-value DeFi applications. Optimization micro-patterns demonstrated different trends: the *Storage Saver* pattern achieved notably higher coverage on Polygon (91.67%) and Fantom (92.64%) compared to Arbitrum

(a) High correlation pairs ($\phi > 0.50$)

Pattern Pair	ϕ
Modifier Usage - Muted	0.58
Operator - Supporter	0.52

(b) Low correlation pairs ($\phi < 0.15$)

Pattern Pair	ϕ
Named Return - Ownable	0.01
Supporter - Emitter	0.03
Implementer - Emitter	0.04

(c) Mutual Exclusivity and Inclusivity by design

Mutually Exclusive	Inclusive (Unidirectional)
Reader ⇔ Operator	Ownable ⇒ Modifier Usage
Emitter ⇔ Muted	Stoppable ⇒ Modifier Usage
Provider ⇔ Supporter	
Multi Return ⇔ Returnless	
Named Return ⇔ Returnless	

Figure 4: Pattern relationships analysis showing: (a) highly correlated pattern pairs, (b) low correlation pattern pairs (excluding zero correlations), and (c) patterns that are mutually exclusive or inclusive by design.

(71.46%), suggesting different prioritization of optimization practices across platforms.

To identify meaningful differences in micro-pattern adoption between blockchains, we employed chi-square tests of independence with Bonferroni correction ($\alpha = 0.05/10 = 0.005$) for multiple comparisons. We complemented this with Cramer's V (w) threshold ($\geq$0.10) to focus on practically significant differences rather than just statistical significance [18].

The statistical analysis revealed that the most substantial differences appeared in five key patterns: *Payable*, *Borrower*, *Delegator*, *Implementer*, and *Storage Saver*. *Storage Saver* showed significant differences across all platform pairs ($p < 0.001$), similar to *Borrower*. *Delegator* pattern showed more nuanced distributions, with one non-significant platform pair (Fantom vs. Polygon, $\chi^2 = 2.21$, $p = 0.137$). While our descriptive statistics suggested variations in security patterns like *Reentrancy Guard*, these differences did not meet our effect size threshold.

The variations in *Storage Saver* patterns could reflect different gas cost structures across platforms, while the distinct *Borrower* usage patterns on Arbitrum ($\chi^2 > 1000$, $p < 0.001$) might indicate platform-specific development practices. However, these hypotheses require further investigation.

(RQ3) How do micro-pattern frequencies vary across different blockchain platforms? While descriptive statistics suggest variations across many micro-patterns, statistical analysis identifies significant platform-specific differences primarily in *Payable*, *Borrower*, *Delegator*, *Implementer*, and *Storage Saver* micro-patterns. This demonstrates our micro-pattern catalog's ability to detect meaningful variations in development practices, though determining the underlying causes requires further research.

6.3 Pattern Relationship Analysis (RQ4)

The relationships between micro-patterns reveal whether our catalog captures distinct design aspects or overlapping concepts. For each entity e and micro-pattern mp, the micro-pattern matching produces a binary outcome $M(mp, e)$ representing micro-pattern presence/absence (see Figure 3). We analyzed these relationships across our dataset of verified smart contracts through correlation metrics that quantify micro-pattern co-occurrences.

The relationship between micro-patterns is quantified using a correlation matrix C of size $|P|\times|P|$, where each element $C(mp_i, mp_j)$ represents the Phi coefficient (ϕ) between micro-patterns mp_i and mp_j. We selected ϕ as it specifically measures correlation between binary variables:

$$\phi(mp_i, mp_j) = \frac{n_{11}n_{00} - n_{10}n_{01}}{\sqrt{n_{1\cdot}n_{0\cdot}n_{\cdot 1}n_{\cdot 0}}}$$

where n_{ij} represents the count of smart contracts with micro-patterns i and j present (1) or absent (0). Based on established guidelines for binary correlation interpretation [5], we considered $\phi > 0.50$ as moderately strong correlation (explaining over 25% of shared variance) and $\phi < 0.15$ as weak correlation (less than 2% shared variance). No patterns exhibited strong correlations ($\phi > 0.635$, explaining over 40% shared variance).

Our analysis revealed predominantly weak correlations between micro-patterns, as reported in Figure 4. Across *all* platforms, only one pair consistently showed moderately strong correlations ($\phi >$ 0.50): *Modifier Usage–Muted* ($\phi \approx 0.54$–0.60). Meanwhile, *Operator –Supporter* ($\phi \approx 0.50$–0.54) attained moderately strong correlations *only* on Ethereum, Polygon, and Optimism. The vast majority of micro-pattern pairs (between 116 to 122 pairs across different chains) showed weak correlations ($\phi < 0.15$).

This independence stems from both intentional design decisions and Solidity's language constraints. For example, *Reader-Operator* micro-patterns are mutually exclusive by design, since they rely on Solidity's `view` and `pure` function types respectively, while *Stoppable* and *Ownable* necessarily include *Modifier Usage*, as they implement their functionality using modifiers. This relationship is unidirectional inclusive, which means even if these micro-patterns require modifiers, the presence of *Modifier Usage* does not imply either *Stoppable* or *Ownable*, as modifiers can be used for other purposes in Solidity smart contracts.

To assess *relationship stability* across platforms, we used Mantel tests [20] to compare *co-occurrence structures* of micro-patterns on different blockchains. These tests yielded low, statistically non-significant correlations ($r = 0.014$–0.145, $p > 0.05$), indicating that how patterns co-occur in one blockchain does not strongly align with how they co-occur elsewhere. Additionally, we computed Spearman correlations [25] (see Figure 5) by flattening each chain's pattern-usage profile into a vector and measuring similarity across platforms; those values also remained relatively low ($r \approx 0.18$–0.28). Together, these findings support our conclusion that micro-pattern co-occurrences remain effectively independent across different blockchains.

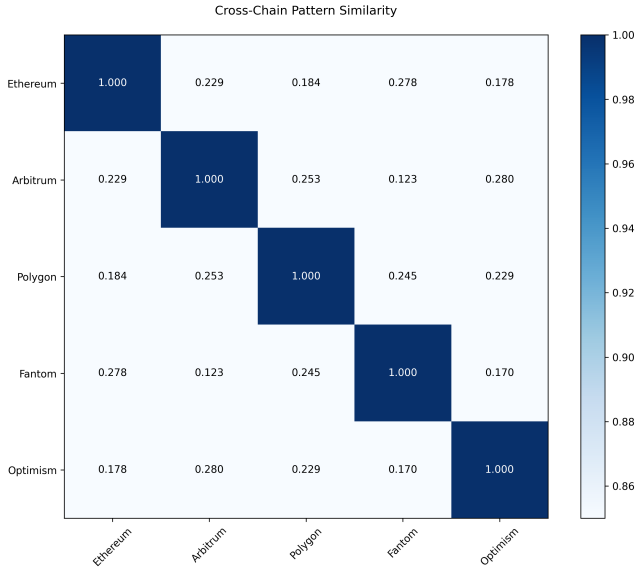

Figure 5: Cross-Chain Pattern similarity

The findings suggested two key insights for smart contract development. First, the widespread weak correlations ($\phi < 0.15$ for >115 micro-pattern pairs per platform) indicated that most micro-patterns represent independent design choices that can be flexibly combined according to specific contract requirements. Second, the few moderate correlations we observed ($\phi > 0.50$) likely reflected inherent relationships in contract design, such as when certain functionality naturally requires specific implementation micro-patterns. This independence across our catalog suggested we identified distinct, non-overlapping aspects of smart contract design.

(RQ4) Which relationships exist between different micro-patterns, and how are they spread? Our analysis showed that most micro-pattern pairs (>115 per platform) were weakly correlated ($\phi < 0.15$), with only a few micro-patterns showing moderate correlations ($\phi > 0.50$). This widespread independence, consistent across all blockchain platforms, suggests our catalog captured distinct, non-overlapping aspects of smart contract design.

7 Threats to Validity

Internal Validity. Our micro-pattern detection framework achieved a 93% success rate in processing 25000 verified smart contracts, with failures primarily stemming from dependency resolution and import remapping issues. While successfully processed contracts demonstrate reliable micro-pattern detection, we acknowledge that the dependency-related failures could have influenced our analysis.

Our focus on publicly verifiable contracts may underrepresent certain patterns, particularly security-focused ones like *Reentrancy Guard*, as security-critical contracts often remain private or use alternative verification platforms. Future work should analyze security-audited contracts from professional sources and audit firms to better understand pattern adoption in high-security contexts.

The framework's core functionality relies on AST parsing and correct dependency resolution, making it sensitive to project structure and compiler versions. To address version compatibility challenges, we implemented a version resolution system that manages multiple Solidity compiler versions. Nevertheless, version-specific language features may affect detection reliability. Initial validation shows promising accuracy across common language features and compiler versions, though future work could expand this validation to cover the full range of Solidity's evolving capabilities.

External Validity. Our dataset spans five public blockchains, ranging from 4,217 (Optimism) to 4,900 (Polygon) smart contracts per chain. This selection was based on their transaction volume, providing representation of mainstream development practices. However, our focus on EVM-compatible chains may limit generalizability to other blockchain architectures. Additionally, findings may not generalize to private/permissioned chains which operate under different security and performance constraints.

Our analysis of the most recent contracts per chain means findings may be influenced by contemporary market conditions, trending DeFi protocols, and popular contract templates. This recent snapshot may not fully represent historical development practices or capture the diversity of contract types across platform lifetimes.

8 Conclusion

This paper introduced the first catalog of 18 micro-patterns for Solidity code, formally defined across five categories: Security, Functional, Optimization, Interaction, and Feedback. Through our automated detection framework, we analyzed 23258 verified smart contracts deployed across five major blockchain platforms (Ethereum, Polygon, Arbitrum, Fantom, and Optimism). Our analysis revealed that 99% of contracts *exhibited at least one micro-pattern*, with an average of 2.76 micro-patterns per contract.

Our findings demonstrated that micro-patterns captured meaningful development practices in Solidity. The *Storage Saver* pattern had the highest adoption with a mean coverage of 84.62%, while security patterns like *Reentrancy Guard* and *Pull Payment* showed more selective implementation, indicating deliberate selection based on specific contract requirements. The statistical analysis revealed significant platform-specific differences, particularly in *Borrower*, *Implementer*, and *Storage Saver* patterns, suggesting that blockchain characteristics may influence development strategies.

The predominantly weak correlations between micro-patterns ($\phi < 0.15$ for over 115 micro-pattern pairs) suggested our catalog captured distinct aspects of smart contract design. This independence across our catalog indicated we identified non-overlapping design aspects in smart contract implementations.

The integration of micro-pattern analysis into the development process can bridge the gap between traditional software metrics and Solidity's unique requirements. The formal definitions and automated detection framework established in this work will provide a foundation for investigating how micro-patterns influence development practices, supporting future research in automated quality assurance for blockchain systems.

References

[1] Andreas M Antonopoulos and Gavin Wood. 2018. *Mastering ethereum: building smart contracts and dapps*. O'reilly Media.

[2] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 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*. Springer, Springer-Verlag, Berlin, Heidelberg, 164–186. https://doi.org/10.1007/978-3-662-54455-6_8

[3] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2020. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)* 53, 3 (2020), 1–43. https://doi.org/10.1145/3391195

[4] Zadia Codabux, Kazi Zakia Sultana, and Byron J Williams. 2017. The Relationship between Traceable Code Patterns and Code Smells. In *SEKE*. Knowledge Systems Institute Graduate School, 444–449. https://doi.org/10.18293/SEKE2017-121

[5] Jacob Cohen. 2013. *Statistical power analysis for the behavioral sciences*. Routledge. 567 pages. https://doi.org/10.4324/9780203771587

[6] Giulio Concas, Giuseppe Destefanis, Michele Marchesi, Marco Ortu, and Roberto Tonelli. 2013. Micro patterns in agile software. In *Agile Processes in Software Engineering and Extreme Programming: 14th International Conference, XP 2013, Vienna, Austria, June 3-7, 2013. Proceedings 14*. Springer, 210–222. https://doi.org/10.1007/978-3-642-38314-4_15

[7] Chris Dannen and Chris Dannen. 2017. Solidity programming. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners* (2017), 69–88. https://doi.org/10.1007/978-1-4842-2535-6_4

[8] Giuseppe Destefanis, Michele Marchesi, Marco Ortu, Roberto Tonelli, Andrea Bracciali, and Robert Hierons. 2018. Smart contracts vulnerabilities: a call for blockchain software engineering?. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, Campobasso, Italy, 19–25. https://doi.org/10.1109/IWBOSE.2018.8327567

[9] Giuseppe Destefanis, Roberto Tonelli, Ewan Tempero, Giulio Concas, and Michele Marchesi. 2012. Micro Pattern Fault-Proneness. In *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, Cesme, Turkey, 302–306. https://doi.org/10.1109/SEAA.2012.63

[10] Vikram Dhillon, David Metcalf, Max Hooper, Vikram Dhillon, David Metcalf, and Max Hooper. 2021. Unpacking ethereum. *Blockchain Enabled Applications: Understand the Blockchain Ecosystem and How to Make it Work for You* (2021), 37–72. https://doi.org/10.1007/978-1-4842-6534-5_4

[11] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*. Association for Computing Machinery, New York, NY, USA, 530–541. https://doi.org/10.1145/3377811.3380364

[12] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, Montreal, QC, Canada, 8–15. https://doi.org/10.1109/WETSEB.2019.00008

[13] Norman E Fenton and Martin Neil. 1999. Software metrics: successes, failures and new directions. *Journal of Systems and Software* 47, 2-3 (1999), 149–157. https://doi.org/10.1016/S0164-1212(99)00035-7

[14] Asem Ghaleb and Karthik Pattabiraman. 2020. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, New York, NY, USA, 415–427. https://doi.org/10.1145/3395363.3397385

[15] Joseph (Yossi) Gil and Itay Maman. 2005. Micro patterns in Java code. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 97–116. https://doi.org/10.1145/1094811.1094819

[16] Sunghun Kim, Kai Pan, and E. James Whitehead. 2006. Micro pattern evolution. In *Proceedings of the 2006 International Workshop on Mining Software Repositories* (Shanghai, China) *(MSR '06)*. Association for Computing Machinery, New York, NY, USA, 40–46. https://doi.org/10.1145/1137983.1137995

[17] Chunmiao Li. 2021. Gas estimation and optimization for smart contracts on ethereum. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1082–1086. https://doi.org/10.1109/ASE51524.2021.9678932

[18] Albert M Liebetrau. 1983. *Measures of association*. Vol. 32. Sage. https://doi.org/10.4135/9781412984942

[19] Stefano Maggioni and Francesca Arcelli. 2010. Metrics-based detection of micro patterns. In *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics* (Cape Town, South Africa) *(WETSoM '10)*. Association for Computing Machinery, New York, NY, USA, 39–46. https://doi.org/10.1145/1809223.1809229

[20] Nathan Mantel. 1967. The detection of disease clustering and a generalized regression approach. *Cancer research* 27, 2_Part_1 (1967), 209–220. https://doi.org/10.1136/bmj.3.5668.473-a

[21] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, San Diego, California, 1186–1189. https://doi.org/10.1109/ASE.2019.00133

[22] Yvonne Murray and David A. Anisi. 2019. Survey of Formal Verification Methods for Smart Contracts on Blockchain. In *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. 1–6. https://doi.org/10.1109/NTMS.2019.8763832

[23] Sidney Siegel. 1957. Nonparametric statistics. *The American Statistician* 11, 3 (1957), 13–19. https://doi.org/10.1080/00031305.1957.10501091

[24] Amritraj Singh, Reza M Parizi, Qi Zhang, Kim-Kwang Raymond Choo, and Ali Dehghantanha. 2020. Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. *Computers & Security* 88 (2020), 101654. https://doi.org/10.1016/j.cose.2019.101654

[25] Charles Spearman. 1961. The proof and measurement of association between two things. *International Journal of Epidemiology* (1961). https://doi.org/10.1037/11491-005

[26] Kazi Zakia Sultana and Byron J. Williams. 2017. Evaluating micro patterns and software metrics in vulnerability prediction. In *2017 6th International Workshop on Software Mining (SoftwareMining)*. IEEE, Urbana, IL, USA, 40–47. https://doi.org/10.1109/SOFTWAREMINING.2017.8100852

[27] Kazi Zakia Sultana, Byron J Williams, and Tanmay Bhowmik. 2019. A study examining relationships between micro patterns and security vulnerabilities. *Software Quality Journal* 27 (2019), 5–41. https://doi.org/10.1007/s11219-017-9397-z

[28] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*. Association for Computing Machinery, New York, NY, USA, 9–16. https://doi.org/10.1145/3194113.3194115

[29] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2021. A Survey of Smart Contract Formal Specification and Verification. *ACM Comput. Surv.* 54, 7, Article 148 (July 2021), 38 pages. https://doi.org/10.1145/3464421

[30] Anna Vacca, Andrea Di Sorbo, Corrado A Visaggio, and Gerardo Canfora. 2021. A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges. *Journal of Systems and Software* 174 (2021), 110891. https://doi.org/10.1016/j.jss.2020.110891

[31] Fernando Richter Vidal, Naghmeh Ivaki, and Nuno Laranjeiro. 2024. Vulnerability detection techniques for smart contracts: A systematic literature review. *Journal of Systems and Software* (2024), 112160. https://doi.org/10.1016/j.jss.2024.112160

[32] Eric W Weisstein. 2004. Bonferroni correction. *https://mathworld. wolfram. com/* (2004).

[33] Maximilian Wohrer and Uwe Zdun. 2018. Smart contracts: security patterns in the ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, Campobasso, Italy, 2–8. https://doi.org/10.1109/IWBOSE.2018.8327565

A Solidity Micro-Patterns Formal Description and Auxiliary Definitions

Table 4: Solidity Micro-Patterns Formalized in First-Order Logic

Pattern	Formal Definition (FOL)
Security	
Ownable	$\exists\, v \in \mathit{stateVars}(C),\ \exists\, m \in \mathit{modifiers}(C),\ \exists\, f \in \mathit{functions}(C) : \big(\mathit{type}(v) = \texttt{address} \land \mathit{isPublic}(v)\big) \land \mathit{checksVar}(m, v) \land \mathit{uses}(f, m)$
Stoppable	$\exists\, v \in \mathit{stateVars}(C),\ \exists\, m \in \mathit{modifiers}(C),\ \exists\, f \in \mathit{functions}(C) : \big(\mathit{type}(v) = \texttt{bool} \land \mathit{isPauseFlag}(v)\big) \land \mathit{checksVar}(m, v) \land \mathit{toggles}(f, v)$
Pull Payment	$\exists, m \in \mathit{stateVars}(C),\ \exists, f \in \mathit{functions}(C) : \big(\mathit{type}(m) = \texttt{mapping(address} \Rightarrow \texttt{uint)}\big) \land \mathit{checksMapping}(m) \land \mathit{transfersToSender}(f)$
Reentrancy Guard	$\exists, v \in \mathit{stateVars}(C),\ \exists, m \in \mathit{modifiers}(C),\ \exists, f \in \mathit{functions}(C) : \big(\mathit{checksVar}(m, v) \implies \mathit{setVar}(m, v) \implies \mathit{execute}(f) \implies \mathit{setVar}(m, v)\big)$
Functional	
Payable	$\exists\, f_i, f_j \in \mathit{functions}(C) : \big((\mathit{name}(f_i) = \texttt{"fallback"}) \land (\mathit{name}(f_j) = \texttt{"receive"})\big)$
Borrower	$\exists\, l \in \mathit{imports}(C),\ \exists\, f \in \mathit{functions}(C) : \big(\mathit{isLibrary}(l) \land \mathit{uses}(f, l)\big)$
Implementer	$\forall, f \in \mathit{functions}(C) : \exists, i \in \mathit{inherited}(C) : \big(\mathit{implements}(f, i)\big)$
Modifier Usage	$\exists\, m \in \mathit{modifiers}(C),\ \exists\, f \in \mathit{functions}(C) : \mathit{uses}(f, m)$
Optimization	
Storage Saver	$\forall\, v \in \mathit{stateVars}(C) : \neg\, \mathit{wasteSpace}(v)$
Reader	$\forall\, f \in \mathit{functions}(C) : \mathit{isView}(f)$
Operator	$\forall\, f \in \mathit{functions}(C) : \mathit{isPure}(f)$
Interaction	
Provider	$\forall\, f \in \mathit{functions}(C) : \mathit{isExternal}(f)$
Supporter	$\forall\, f \in \mathit{functions}(C) : \mathit{isInternal}(f)$
Delegator	$\exists\, c \in \mathit{stateVars}(C),\ \exists\, f \in \mathit{functions}(C) : \big(\mathit{isContract}(\mathit{type}(c)) \land \mathit{delegatesTo}(f, c)\big)$
Feedback	
Named Return	$\forall\, f \in \mathit{functions}(C) : \mathit{hasNamedReturns}(f)$
Returnless	$\forall\, f \in \mathit{functions}(C) : \mathit{returnCount}(f) = 0$
Emitter	$\forall\, f \in \mathit{functions}(C) : \mathit{hasEvent}(f)$
Muted	$\forall\, f \in \mathit{functions}(C) : \neg\, \mathit{hasEvent}(f)$

Contract Accessors	
$\mathit{stateVars}(C)$	Returns the set of state variables in C.
$\mathit{functions}(C)$	Returns the set of functions in C.
$\mathit{modifiers}(C)$	Returns the set of custom modifiers in C.
$\mathit{enums}(C)$	Returns the set of enums declared in contract C.
$\mathit{imports}(C)$	Returns the set of libraries or external contracts imported by C.
$\mathit{storageSlots}(C)$	Returns the set of storage slots used by C.

Function/Variable Properties	
$\mathit{isPure}(f)$	True if f is declared `pure`.
$\mathit{hasExternalCall}(f)$	True if f performs an external call.
$\mathit{isView}(f)$	True if function f is declared `view` (read-only).
$\mathit{returnCount}(f)$	Number of return values in f.
$\mathit{hasNamedReturns}(f)$	f has named returns.
$\mathit{overridesAbstract}(f)$	f overrides an abstract method.
$\mathit{type}(v)$	Returns the type of variable v.

Checks and Delegation	
$\mathit{toggles}(f, v)$	True if f toggles (switches) a bool variable v.
$\mathit{delegatesTo}(f, c)$	True if f delegates calls to contract c.
$\mathit{checksVar}(m, v)$	True if modifier m checks the value of variable v.
$\mathit{checksMapping}(m, v)$	True if modifier m checks mapping variable v for permissions.
$\mathit{checksLock}(m, v)$	True if modifier m checks a reentrancy lock variable v.

Boolean Flags/Storage	
$\mathit{wasteSpace}(v)$	True if v can be placed in a previously non-full slot.
$\mathit{size}(s)$	Byte size of storage slot s.
$\mathit{isLibrary}(l)$	True if l is a Solidity library.
$\mathit{isPauseFlag}(v)$	True if v is a boolean flag for a paused/stopped state.
$\mathit{isLock}(v)$	True if v is used as a reentrancy lock.
$\mathit{transfersEther}(f)$	True if function f transfers Ether.
$\mathit{hasEvent}(f)$	True if function f emits an event.

A more exhaustive set of definitions is provided in our replication package, including visibility checks, event-related properties, and additional internal predicates.

Figure 6: Representative Subset of Auxiliary Functions and Predicates