Demystifying the Composition and Code Reuse in Solidity Smart Contracts

Kairan Sun
Nanyang Technological University
Singapore, Singapore
sunk0013@e.ntu.edu.sg

Zhengzi Xu*
Nanyang Technological University
Singapore, Singapore
zhengzi.xu@ntu.edu.sg

Chengwei Liu
Nanyang Technological University
Singapore, Singapore
chengwei001@e.ntu.edu.sg

Kaixuan Li
East China Normal University
Shanghai, China
kaixuanli@stu.ecnu.edu.cn

Yang Liu
Nanyang Technological University
Singapore, Singapore
yangliu@ntu.edu.sg

ABSTRACT

As the development of Solidity smart contracts has increased in popularity, the reliance on external sources such as third-party packages increases to reduce development costs. However, despite the use of external sources bringing flexibility and efficiency to the development, they could also complicate the process of assuring the security of downstream applications due to the lack of package managers for standardized ways and sources. While previous studies have only focused on code clones without considering how the external components are introduced, the compositions of a smart contract and their characteristics still remain puzzling.

To fill these gaps, we conducted an empirical study with over 350,000 Solidity smart contracts to uncover their compositions, conduct code reuse analysis, and identify prevalent development patterns. Our findings indicate that a typical smart contract comprises approximately 10 subcontracts, with over 80% of these originating from external sources, reflecting the significant reliance on third-party packages. For self-developed subcontracts, around 50% of the subcontracts have less than 10% unique functions, suggesting that code reuse at the level of functions is also common. For external subcontracts, though around 35% of the subcontracts are interfaces to provide templates for standards or protocols, an inconsistency in the use of subcontract types is also identified. Lastly, we extracted 61 frequently reused development patterns, offering valuable insights for secure and efficient smart contract development.

CCS CONCEPTS

• **General and reference → Empirical studies**; • **Software and its engineering → Software libraries and repositories**.

KEYWORDS

smart contract composition, code reuse, development pattern

*Zhengzi Xu is the corresponding author

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3616270

ACM Reference Format:
Kairan Sun, Zhengzi Xu, Chengwei Liu, Kaixuan Li, and Yang Liu. 2023. Demystifying the Composition and Code Reuse in Solidity Smart Contracts . In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3611643.3616270

1 INTRODUCTION

Smart contracts are a special type of program built on top of blockchain technology to automate, verify, and enforce the negotiation of agreements between parties. The development of smart contracts has gained significant traction in recent years as it has the potential to revolutionize a wide range of industries such as financial services, supply chain management, and the Internet of Things (IoT) [23]. To lighten the development of smart contracts, Solidity [1] has been proposed and widely used as the primary language in mainstream blockchain platforms. For instance, Ethereum [18], as one of the most popular open-source blockchain platforms, has embraced the diverse ecosystem of decentralized applications (DApps) by using Solidity smart contracts.

However, as the complexity of DApps grows, the development of smart contracts (i.e., smart contracts in this paper here and below refer to Solidity smart contracts) is increasingly relying on well-established subcontracts (i.e., contract-level code blocks as detailed in Section 2) from third parties to avoid re-inventing wheels and reduce development efforts. While the reuse of third-party subcontracts also poses a new threat that, like supply chain attacks in traditional Web2 ecosystems, vulnerabilities and flaws from third-party contracts could further collapse downstream applications, especially in the context of early-staged smart contract development that lacks mature dependency management solutions. As reported in Feb 2022 [14], two critical vulnerabilities in the subcontracts provided by Multichain [11] affected 7,962 user addresses, and a total of over 3 million dollars were exploited. In this case, disclosing the potential security risks hidden in reused third-party subcontracts in time is vital to the assurance of Web3 Security.

Nevertheless, the diverse and flexible third-party subcontract reuse makes it non-trivial to precisely manage third-party subcontracts introduced during smart contract development. Although most commonly used third-party subcontracts are released in the format of NPM packages, and they can be introduced by NPM

dependencies, there are still various ways and practices to reuse existing third-party subcontracts (i.e., simply declaring dependencies by URLs from Github [2] Repos, IPFS gateways [3], Swarm gateway [4], etc.). Despite the convenience, such diversity and flexibility also complicate the security assurance of downstream applications since the URLs could be untrustworthy. Moreover, though NPM offers version management of dependencies, it is challenging to trace the versions of the imported subcontracts, especially after on-chain. This makes it difficult to derive standardized solutions to identify, manage and remediate known vulnerabilities existing in dependencies like what Software Composition Analysis tools [48] do to Web2 applications. In addition to external imports, many developers also tend to directly clone the existing subcontracts into their contracts for convenience. As revealed by Pierro et al [41], 79.1% of smart contracts contain duplicated code. Such code clones further challenge the management of reused third-party subcontracts.

To unveil the mysteries in the development of smart contracts, various exploratory studies have been conducted. Some researchers investigated the characteristics of existing smart contracts such as the design patterns [25] and source code complexity [40] of different kinds of smart contracts, while some researchers [34, 35] studied the code reuse of smart contracts at different granularity including contracts, subcontracts, and functions. However, most existing works only focused on a single granularity of smart contracts such as subcontracts or functions, neglecting how these elements are introduced during development. This may limit the understanding of smart contracts and lead to incomplete or inaccurate conclusions about smart contracts. The lack of exploration in such compositions also compromises the derived guidance on further management and governance of the reuse of smart contracts across the ecosystem. To fill these gaps, we conducted an empirical study to investigate the reuse of the subcontracts from the perspective of contract decomposition on the Ethereum blockchains.

The overall framework of this study is illustrated in Figure 1. We first collected over 350,000 smart contracts deployed between January 2021 and January 2023 from Etherscan [7], a leading blockchain explorer and analytic platform for Ethereum. Additionally, to enhance our understanding of the existing third-party packages for smart contract development and expand our dataset further for subsequent code reuse analysis, a list of 353 third-party packages from NPM [9] and GitHub [2] Repos was compiled from the import statements of the collected contracts. To ensure the robustness of our subsequent code reuse analysis, we first validated the contracts and parsed them into JSON files with ANTLR-4 [6]. We then examined and filtered out contracts with no or slight modification by conducting a contract-level code clone detection. After this process, 296,236 distinct contracts are sorted out for further studies. Our empirical study offers a comprehensive view of contract composition and code reuse practices considering the regulations to introduce external components. Specifically, we first decompose contracts into various subcontracts based on their origins and how they were introduced into the contracts. This allows us to separate self-developed and external subcontracts for in-depth code reuse analysis. For self-developed subcontracts, we examine function-level code reuse, summarizing the commonly reused functions and functions required to be self-defined. For external subcontracts, we study the usage of frequently reused subcontracts and clones

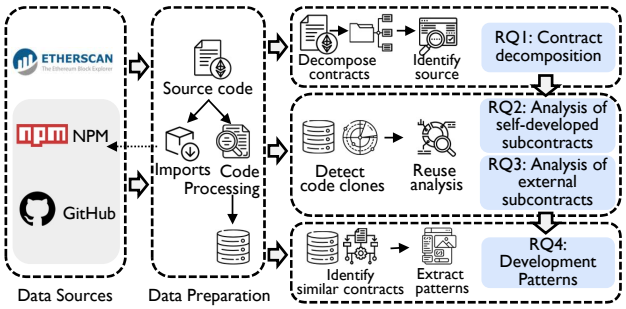

Figure 1: Framework of the study

across third-party packages. Lastly, based on this comprehension, we extract and analyze the frequently reused development patterns.

Throughout the empirical study, we concluded some findings as follows. For example, ① More than 80% of the subcontracts are from external sources. The largest identified external source is NPM, accounting for over 72% of the total external subcontracts. ② Despite the fact that Solidity allows to include NPM dependencies by specifying the required subcontract with the package name in import statements, over 56% of the cloned subcontracts are sourced from NPM packages, indicating a mess in dependency management. ③ The code reuse of functions is common since around 50% of self-developed subcontracts have less than 10% unique functions. As the number of functions increases in a self-developed subcontract, the percentage of unique functions tends to decrease. This highlights the tendency of developers to rely on pre-existing code blocks when developing smart contracts. ④ Though around 35% of the external subcontracts are interfaces, commonly used as protocol templates, we identified an inconsistency in the use of subcontract types which presents challenges in contract management. ⑤ We identified 61 prevalent development patterns, 68.75% of which are for token creations. While these patterns could greatly reduce development costs, it would be risky if a vulnerability or security exploit was found in the patterns. Hence, management in such patterns to continuously monitor and assess their security is important.

In conclusion, the main contributions of this study are as follows.

- We provide a thorough analysis of common approaches to introducing subcontracts in smart contracts, with a further code reuse analysis to identify the most frequently reused code blocks and their purpose.
- We conduct a comprehensive study of 353 identified third-party packages for smart contract development, analyzing their usage frequency and functional properties.
- We summarize and assess 61 frequently used development patterns for smart contracts, which could be potentially used in low-code development.
- Based on our results and findings, we provide an in-depth discussion regarding the challenges in managing external components and development patterns for smart contracts.

2 BACKGROUND

In this section, we provide explanations and definitions of terminologies that have been used in this study.

✦ **Smart Contract and Subcontract.** In this study, a smart contract refers to code blocks sharing the same address while a subcontract refers to a contract-level code block. According to Solidity documentation [12], four types of subcontracts have been defined based on the way they are used:

- ***interface***: subcontracts to define a standard or protocol without methods implementations.
- ***abstract***: subcontracts to define a basic structure with some methods implementations.
- ***contract***: complete and executable subcontracts.
- ***library***: subcontracts to provide reusable code blocks for common operations without storage.

To clarify this further, in our study, subcontract refers to interfaces, abstract contracts, contracts, and libraries as detailed above. This serves to distinguish between modularized contract-level code blocks (subcontracts) and the composite code eventually deployed on the blockchain (contract). Essentially, these subcontracts can be combined to form a complete contract ready for deployment.

✦ **Ethereum [18] and Etherscan [7].** Ethereum is an open-source blockchain platform supporting decentralized applications through smart contracts. For our study, we selected two networks:

- ***Ethereum Mainnet***: the main blockchain network for Ethereum blockchain involving real transactions.
- ***Goerli Testnet***: a testing network for Ethereum blockchain to be used for smart contract testing and developing.

Etherscan is a blockchain explorer for the Ethereum blockchain to provide detailed information on blocks including transactions, addresses, and the source code of smart contracts if available.

✦ **EIPs.** EIPs (Ethereum Improvement Proposals) [8] are formal proposals for the Ethereum network that help to document the standardized protocols with possible implementations in a well-organized format. As EIPs are audited and expected to be safe to reuse, we included EIPs as a basic approach to evaluating the safety of smart contracts, third-party packages, and development patterns.

✦ **Clone Types.** Four clone types categorize code clones in software systems [42]. In our study, we focus on type-2 clones, which are syntactically identical codes with different identifiers or literals, and use type-1 clones, exact code copies differing only in whitespace or comments, for validation. We exclude type-3 and type-4, which concentrate on major changes and semantic parallels. While smart contracts are relatively simple and standardized, type-3 and type-4 clones may introduce a significant amount of false similarities, affecting the overall accuracy of the result.

3 EMPIRICAL STUDY

3.1 Research Questions

The structure of our research questions is presented in Figure 2. To explore contract compositions and common development practices with considerations of dependency introduction, we first decompose contracts into subcontracts, identifying their origins and introduction methods. We then conduct in-depth code reuse analyses on subcontracts from various sources separately (i.e., self-developed or external), providing insights into their individual characteristics. Lastly, with a better understanding of the compositions of contracts and code reuse, we extracted frequently used development patterns.

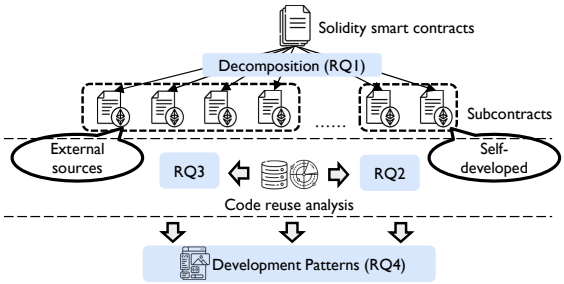

Figure 2: Relation between research questions

3.1.1 ***Contract Clone Decomposition (RQ1): What are the common approaches to introducing subcontracts into smart contracts and their sources?*** Identifying contract compositions and quantifying the use of various approaches to introducing subcontracts. The results obtained from the exploration serve as the fundamental for the subsequent analysis.

3.1.2 ***Analysis of Self-Developed Subcontracts (RQ2): What are the characteristics of self-developed subcontracts?*** Understanding the characteristics and extent of code reuse in self-developed subcontracts can highlight common areas of customization, thereby informing tailored development practices and tooling.

3.1.3 ***Analysis of External Subcontracts (RQ3): What are the characteristics of external subcontracts?*** Analyzing the characteristics and common applications of top reused external subcontracts and third-party packages to aid in better contract development guidance and third-party package management.

3.1.4 ***Development Patterns (RQ4): What are the commonly used development patterns in smart contract development?*** Extracting and analyzing frequently reused development patterns in smart contract development which reflects common usage and safety practices, thus informing best practices and potentially influencing the design of safer smart contract development frameworks.

3.2 Dataset Preparation

An overview of our data preparation process is presented in Figure 3. We collect smart contract source code from Etherscan [18], NPM [9], and GitHub Repos [2]. To normalize and transfer the collected code into a suitable format for the subsequent analysis, we also conduct a series of data preprocessing before our analysis.

3.2.1 ***Contract Collection.*** For our empirical study, we first collected the source code of 221,309 deployed smart contracts on the Ethereum Mainnet and 131,207 contracts on the Goerli Testnet [15] from Etherscan [7]. To extend our dataset and better trace subcontract origins, we then collected the source code of third-party packages for smart contract development, the list of which was compiled by performing a syntax-based extraction to the import statements from the primary dataset. In the end, we managed to identify the use of 337 NPM packages and 33 GitHub repositories that are used in smart contract development.

3.2.2 ***Contract Preprocessing.*** After collecting the source code of smart contracts, the following three steps are taken in sequence to process the contracts for the subsequent experiments.

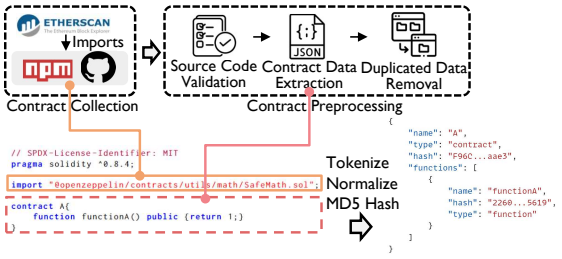

Figure 3: Overview of data preparation

Source Code Validation. As our primary contract collection is from Etherscan [7], an open platform where users can upload source code, we conducted a two-stage validation to ensure the integrity and completeness of the dataset. First, we confirm the contracts are stored in a single file with all required meta details (i.e., address, contact name, and creation date). Second, we verify each contract is non-empty by inspecting the content within the outermost curly brackets of each subcontract. After the validation, approximately 3% of the contracts are filtered out, comprising around 6,241 on the Ethereum Mainnet and 2,240 on the Goerli Testnet.

Contract Data Extraction. To ensure an accessible representation of contract data for further analysis, we developed a parser for Solidity smart contracts using ANTLR4 [6]. The parser first converts smart contracts into token streams, extracting two types of information for each token: its original text and its token type identifier. We then extract code blocks such as subcontracts and functions from the contract and compute the MD5 hash for each block after type-2 normalization (i.e., remove white spaces and comments, and rename variables) to facilitate further analysis. After this, each contract will be saved as a JSON file containing all the extracted and computed information as depicted in Figure 3. An example of the complete output file can be found on our website [17].

Duplicated Contract Removal. The aim of this study is to determine the common practices in smart contract development. However, having multiple copies of the duplicated contracts with only minor changes or no changes may skew the results. To address this, we grouped contracts based on their contract-level MD5 hashes after type-2 normalization. This ensures that contracts within the same group can only have minor differences such as variable names. As a result, 7,617 groups were identified to have more than one contract. We then conducted a manual inspection for the groups and confirmed that 90.17% are created by the same teams as detailed in Section 3.3. Hence, to avoid the overrepresentation of similar contracts and enhance the diversity of our datasets, we decided to retain only the earliest version of each contract group.

In the end, we kept 189,229 distinct contract entries for Ethereum Mainnet and 107,034 entries for Goerli Testnet. The processed data were saved to a database for use in later code reuse analysis.

3.3 Contract Clone Decomposition (RQ1)

3.3.1 ***Experiments Setup.*** To gain insights into the compositions of contracts and common practices in development, we commenced our study by assessing contract-level code clones and quantifying the various approaches to introducing subcontracts into contracts. This lays the foundation for the subsequent analysis.

Initially, for the contract-level clone analysis, we examined the duplicated contract groups identified during data preprocessing as detailed in Section 3.2.2. A team of three smart contract auditors conducted a manual inspection on a randomly selected sample with 366 groups. The sample size was determined by Cochran's equation [29] together with a population correction [36]. During the inspection, each auditor independently evaluated the reason behind duplication. Collective discussions were then held until an agreement was reached in cases where initial opinions varied.

Upon obtaining a comprehensive understanding of contract clones, we directed our focus toward compositions of contracts. Recognizing that Etherscan [7] includes externally imported subcontracts in the published contract source code, we integrated the approaches to introducing external subcontracts into our analysis. Upon identifying subcontract clones based on the pre-computed MD5 hashes, we categorized them by their introduction approaches, quantified the prevalences of these approaches, and traced the origins of externally introduced subcontracts.

3.3.2 ***Results and Discussion.*** Our findings for contract and subcontract clones are detailed separately in the following subsections.

Analysis of Contract Clones. The contract clone ratio in Ethereum Mainnet and Goerli Testnet accounts for 14.50% and 18.42% respectively. Out of these, a total of 7,617 contract groups in Ethereum Mainnet were identified as duplicates during the data preprocessing stage (i.e., contracts within each group have the same MD5 hash). Upon conducting a manual inspection of the sample groups, we discovered that about 89.16% of these groups are caused by version iterations while 3.31% are linked to the use of generators. The rest groups also seem to follow predefined templates, however, the exact reason behind this still remains unclear due to the lack of explicit statements or available online documentation. On the other hand, 90.17% of the groups consist of contracts deployed by the same team or individuals. This was verified by investigating the record of the author in the source code and the details of the contract creator on Etherscan [7]. As a result of the inspections, we could confirm that the identified groups do not contribute to the diversity of contract development, hence, it is acceptable to be excluded from our analysis to avoid an overrepresentation of similar contracts.

To gain more insights into the patterns and characteristics of contract cloning, we analyzed the distribution of group size. Despite the high volume of identified groups, only 2 groups contain over 1,000 contracts, with mere 40 groups exceeding 100 contracts. This indicates that high-volume duplications are uncommon in the dataset. In fact, such extensive duplications are mostly linked to the use of token generators. For instance, the 2 groups with more than 1,000 contracts are both the products of token generators. The largest group with a size of 2,849 is generated by the ERC1155 token creator published by manifoldxyz [20] while the second largest group with a size of 1,105 is due to the use of the ERC20 token generator published by [30]. Apart from that, we noticed the existence of a type of contract that is created and executed on demand with only minor changes. An example of this contract type is lockEtherPay, which is a token lock that only changes the address of the beneficiary per deployment, which is discussed by Kondo et al. [35]. On the other end of the spectrum, we observed that groups due to version iterations usually have fewer than 5 contracts.

Table 1: Average percentage of subcontracts that in introduced using various approaches

	Self-Developed	Imported	Cloned
Ethereum Mainnet	16.43%	22.62%	60.95%
Goerli Testnet	17.68%	39.82%	42.51%

Moreover, we also quantified the purpose of the groups of contracts during the inspection. Over 80% of the groups are for tokens with various functionalities. Specifically, 66.10% of the groups are for ERC20 tokens and 22.38% of the groups are for ERC721 tokens.

Finding 1: Among the 7,617 groups of duplicated contracts, over 80% of the groups are related to tokens, with ERC20 tokens making up 66.10% of the groups. Large-scale contract duplications usually come from using generators. Additionally, 90.17% consist of contracts created by the same team.

Analysis of Subcontract Clones. A smart contract includes around 10 subcontracts on average. These subcontracts can be classified into three categories based on how they are introduced to a contract: ① **Imported subcontract.** A subcontract that is introduced by an import statement referring to an external source such as an NPM package or a URL. ② **Cloned subcontract.** A subcontract that is copied from another external source including earlier deployed smart contracts and third-party packages without an import statement referring to the source. ③ **Self-developed subcontract.** A subcontract that is neither imported nor cloned from external sources including earlier deployed smart contracts and third-party packages. Among the three categories, ① and ② are considered external subcontracts. In our case, we first confirmed imported subcontracts in each contract by checking its import statements. Next, we conducted a subcontract-level code clone detection to identify cloned subcontracts. In the end, we classified those not imported or cloned as self-developed subcontracts.

To better understand the compositions of contracts, we then determined the average distribution of the aforementioned three types of subcontracts within smart contracts for both Ethereum Mainnet and Goerli Testnet, as shown in Table 1. The result indicates that cloning is the most prevalent approach to introduce subcontracts in smart contract development, especially for contracts on Ethereum Mainnet, where 60.95% of the subcontracts are cloned. Additionally, fewer than 20% of the subcontracts are classified as self-developed, which suggests that the current development of smart contracts heavily relies on external solutions. Though the use of external solutions helps to reduce development costs and improve code quality, it may also introduce potential security risks and make it harder to maintain the code in the long run.

Furthermore, we managed to trace the origins of some external subcontracts introduced via different approaches. For imported subcontracts, we focused on the two types of import statements to introduce subcontracts from external sources as documented in [5]: ① Import from NPM packages using import statement followed by the name of the NPM package; ② Import from URLs using import statement followed by a URL such as Github [2], IPFS gateway [3], and Swarm gateway [4]. In our dataset, we only discovered the use of URLs referring to GitHub Repos which accounts for 0.13%

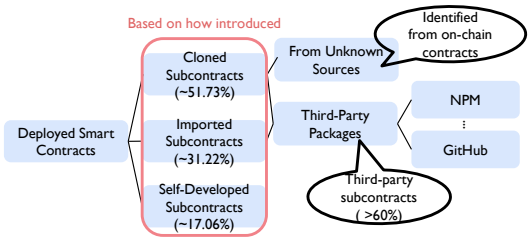

Figure 4: Decomposition of deployed smart contracts

of imported subcontracts, indicating an imbalance in the use of the two types of import statements in smart contract development. In fact, about 51% of the identified GitHub Repos have published NPM packages. This suggests that NPM acts as the package manager for most third-party packages in smart contract development. However, NPM is not optimized for managing Solidity libraries or dependencies. For example, ① Lack of versioning and deployment management. NPM does not help with the versioning of subcontracts and deployment of smart contracts in various environments. Important information such as the version of imported subcontracts may be lost after the contract is published online as the log file generated by NPM to record such information will not be published together. ② Lack of security. NPM does not provide necessary security measures for managing Solidity libraries, which is crucial for smart contracts to ensure the reliability of on-chain information such as the integrity of DApps and the protection of valuable assets. ③ Lack of compatibility. NPM was originally designed for managing libraries, and as a result, it does not directly compatible with Ethereum. To use NPM packages in Solidity smart contracts, extra steps are needed, such as specifying the path of the imported subcontract within the NPM package in the import statement.

While for cloned subcontracts, we discovered that over 56% of the cloned subcontracts are sourced from third-party packages in both Ethereum Mainnet and Goerli Testnet. This indicates a common practice of reusing subcontracts from third-party packages by cloning instead of using import statements. In the view of developers, some possible reasons for this could be: ① Lack of awareness. Developers may lack awareness or familiarity with available package management tools. ② Lack of trust. Developers may lack trust in existing package management tools for smart contract development, hence prefer to copy and paste directly from their reliable sources. ② User-unfriendly. Developers may find it difficult to identify the path of a subcontract within a third-party package which is needed by the import statements in Solidity. ③ Need for customization. Developers may want to make modifications to the subcontract, which is not allowed by using import statements.

Finding 2: We summarized the composition of smart contracts in Figure 4. Smart contracts often consist of three types of subcontracts: cloned (51.73%), imported (31.22%), and self-developed (17.06%). However, over 56% of the cloned subcontracts can be imported from third-party packages. Additionally, we found that NPM contains most third-party packages for smart contract development, though its use is not optimized.

Table 2: Top reused external subcontracts

Rank	Known Source						Unknown Source					
	Ethereum Mainnet			Goerli Testnet			Ethereum Mainnet			Goerli Testnet		
	Name	Kind	%	Name	Kind	%	Name	Kind	%	Name	Kind	%
1	IERC20	interface	39.66%	Context	abstract	25.75%	IUniswapV2Factory	interface	25.79%	ERC721Creator	contract	5.30%
2	Context	abstract	34.44%	IERC165	interface	22.36%	SafeMath	library	22.20%	VRFCoordinatorV2Interface	interface	1.26%
3	IERC165	interface	28.47%	IERC721Receiver	interface	16.70%	Context	abstract	21.76%	ERC721Creator	contract	1.03%
4	IERC721Receiver	interface	25.04%	ERC165	abstract	15.12%	IUniswapV2Router02	interface	12.58%	IFactory	interface	0.97%
5	ERC165	abstract	21.90%	Address	library	14.11%	Ownable	contract	12.39%	SafeMath	library	0.64%
6	IERC721Metadata	interface	20.51%	IERC721Metadata	interface	12.63%	ERC721Creator	contract	10.51%	Context	abstract	0.58%
7	Strings	library	20.39%	Ownable	abstract	11.28%	IUniswapV2Router02	interface	8.70%	ERC1155Creator	contract	0.48%
8	IERC721	interface	17.32%	IERC20	interface	10.55%	Ownable	contract	7.19%	IERC20	interface	0.40%
9	Address	library	17.08%	IERC721	interface	10.30%	SafeMathUint	library	4.28%	ContextMixin	abstract	0.36%
10	Ownable	abstract	16.27%	Strings	library	9.19%	SafeMathInt	library	4.20%	PriceConverter	library	0.35%

(1) The % in the table represents the proportion of use of the subcontract out of total contracts.

3.4 Analysis of Self-Developed Subcontracts (RQ2)

3.4.1 ***Experiments Setup.*** To assess code reuse in self-developed subcontracts, we implemented a two-step analysis. First, we conducted a function-level clone detection within the subcontract to determine the extent of code reuse in contract development. Second, we quantified the percentage of unique functions in each subcontract, providing insights to understand developers' behavior of code reuse in relation to the complexity of the contract.

Additionally, to comprehend the common usage of the reused functions, we quantified these functions based on their functionality by identifying keywords in function names (e.g., mint and transfer), the origins of functions if identifiable, and function types (i.e., *function*, *modifier*, *event*, and *constructor*). We also conducted case studies for the most frequently reused functions to better understand the trends of function-level code reuse.

3.4.2 ***Results and Discussion.*** The result for function reuses in self-developed subcontracts is presented in Figure 5 and Figure 6.

Analysis of Function Clones. The result reveals consistent function reuse patterns for self-developed subcontracts across Ethereum Mainnet and Goerli Testnet as suggested in Figure 5. Approximately 50% of the identified self-developed subcontracts contain less than 10% unique functions, denoting that the reuse of code at the level of functions is also a common practice in smart contract development. Despite this, while cataloging function usage frequency within self-developed subcontracts, we found that only 27.10% and 28.05% of functions were reused for Ethereum Mainnet and Goerli Testnet respectively. This indicates that the commonly reused functions tend to be concentrated on a small set of functions.

Furthermore, in exploring the correlation between the number of unique functions and the total function count in self-developed subcontracts, we found self-developed subcontracts with fewer functions have a higher ratio of unique functions, as depicted in Figure 6. Particularly, self-developed subcontracts with less than 10 functions showed the highest unique function ratio (40.53%). 72 self-developed subcontracts are identified to have 100% unique functions, 56.94% of which only have one function. However, as more functions are required in self-developed subcontracts, the percentage of unique functions drops. This indicates that developers tend to reuse more code blocks from external sources during development rather than writing everything from scratch when the required functionality becomes more complicated. While such

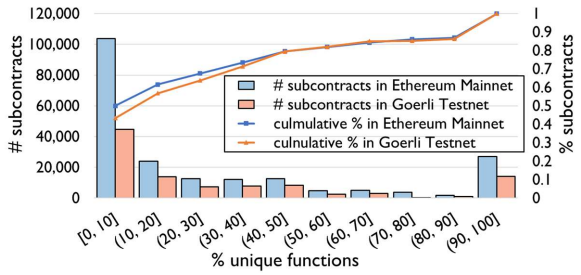

Figure 5: Unique functions in self-developed subcontracts for both Ethereum Mainnet and Goerli Testnet

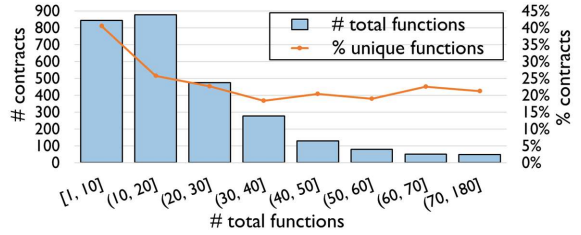

Figure 6: Average percentage of unique functions with respect to the total functions in self-developed subcontracts in smart contracts

code reuse is usually to mitigate potential vulnerabilities, the safety of external functions may not be guaranteed since only around 8% are from well-established packages. Additionally, integrating these functions within the subcontract could be challenging, especially when interactions between functions need to be considered.

> **Finding 3:** Code reuse is a common practice at the function level, especially when the required functionality is complicated. Additionally, we noticed consistent reuse of a small set of functions across subcontracts, reflecting common functional requirements during development.

Analysis of Function Usage. The majority of unique functions and reused functions are standard functions. A substantial portion of unique functions is constructors, taking around 21.56% in Ethereum Mainnet and 13.64% in Goerli Testnet. Constructors are a special type of function that is called during contract creation to initialize the contract and set the initial state. This implies that the

contract initialization process often required specific customization to cater to the particular needs of individual smart contracts.

On the other hand, reused functions often represent commonly needed functionalities. We found that over 20% of these functions are getters and setters, which are widely used to access and modify the current state of the contract. Functions related to minting and transferring also make up a significant portion of both reused and unique functions, accounting for more than 10% in total.

In fact, the balance between unique and reused functions in smart contracts appears to be a blend of custom-tailored operations and widely used standard functionalities. As smart contract development continues to evolve, understanding the interplay and appropriate application of these unique and reused functions will be crucial in optimizing contract efficiency and effectiveness.

Finding 4: Both unique and reused functions are primarily standard functions, with constructors constituting a notable portion of unique functions, indicating frequent customization requirements during contract initialization.

3.5 Analysis of External Subcontracts (RQ3)

3.5.1 ***Experiments Setup.*** To comprehend the significance and implications of external subcontracts, which account for 80% of the total, we conducted experiments from three aspects. ① **Analysis of subcontract type.** To understand common usages of external subcontracts, we quantified the use of the four types of subcontract that are defined for various usage (i.e., *contract*, *abstract*, *interface*, and *library*). Furthermore, we examined the inconsistency in the use of subcontract types. ② **Analysis of subcontract usage frequency.** To gain insights into the trends of external subcontract utilization, we ranked the subcontracts based on their frequency across our dataset and then conducted case studies with the most reused contract examples. ③ **Overview of third-party package.** To comprehend the usage of existing third-party packages, we analyzed package usage frequency and uncovered potential code clones across packages by conducting a subcontract-level clone detection on all recognized third-party packages.

3.5.2 ***Results and Discussion.*** We present our findings for the three stages in the three subsections below separately.

Analysis of Subcontract Type. The usage frequency of these four types of subcontracts is detailed in Table 3. Among the four types of subcontracts, *interface* takes the most percentage in both Ethereum Mainnet (36.59%) and Goerli Testnet (34.85%). This is reasonable since *interface* is usually used as the template of well-established standards (e.g., EIPs) and protocols (e.g. Uniswap [19]). Hence, the use of *interface* often indicates reusing established standards and protocols to reduce development costs while ensuring consistency and interoperability between contracts.

While examining external subcontracts with the type of *contract*, which account for over 30% of the total subcontracts in both networks, we observed an inconsistent use of subcontract types. One case study we provide in this paper is regarding SafeMath, a utility library for common safe math operations. In the context of smart contract development, *library* refers to a reusable code block for multiple contracts to perform common tasks without storage while *contract* is a self-contained unit of logic to execute its own functions with storage. Hence, SafeMath is expected to be defined as *library* here. However, it was incorrectly defined as *contract* in 10% of the over 26,000 definitions. We manually checked the subcontracts defined as *contract* and noticed the declarations that SafemMath is a library in code comments. Similarly, IERC20 is meant to be an *interface* to provide the definition of EIP20 [13] but was defined as *contract* without function implementations in some contracts. In fact, over 93% of the identified cases involve the type of *contract*.

Table 3: Percentage of external subcontracts for various types

	Interface	Contract	Abstract	Library
Ethereum Mainnet	36.59%	34.86%	16.11%	12.45%
Goerli Testnet	34.85%	34.28%	16.99%	13.88%

These inconsistencies could stem from developer confusion regarding the usage of subcontract types, potentially leading to compatibility issues, difficulties in maintenance, and increased vulnerability risks. The flexible usage of *contract* type may contribute to this confusion since it can mimic the functionalities of *interface*, *abstract*, and *library*. In fact, over 93% of the identified cases involve the type of *contract*. Though various types of subcontracts are expected to help developers to organize code more efficiently while improving the maintainability and readability of code blocks, the inconsistency in the use of subcontract types brings more challenges to smart contract management.

Finding 5: While the 35% of external subcontracts to be *interface* indicates a good practice to follow well-established standards and protocols, the usage of subcontracts with the type of *contract*, accounting for another 35% of the total subcontracts, reveals the inconsistency in the use of subcontract types, which brings challenges to smart contract management.

Analysis of Subcontract Usage Frequency. The result of top-reused external subcontracts is presented in Table 2. Here, by subcontracts with known sources, we refer to imported subcontracts and cloned subcontracts sourced from our collected third-party packages. While subcontracts with unknown sources are cloned subcontracts with their origins not confirmed yet. Due to the page limit, we only provide the top 10 reused subcontracts in this paper while the rest will be available on our website [17].

The result suggests that third-party subcontracts are used more frequently compared to those from unknown sources, particularly in Goerli Testnet where the most frequently reused subcontract from unknown sources only appears in 5.3% contracts, which is lower than the 10th most frequently reused third-party subcontract. Moreover, the frequent use of IERC20 and IERC721 in third-party subcontracts highlights that one of the most popular applications in smart contract development is the creation of tokens including both fungible (ERC20) and non-fungible (ERC721) tokens.

In addition, though subcontracts with the type of *contract* take around 35% in external subcontracts as shown in Table 3, most of them are not frequently reused subcontracts. In the top 1,000 reused external subcontracts, subcontracts with the types of *abstract* and *library* make up 50% of the total while *interface* takes another 32%. This suggests a good practice in smart contract development since subcontracts with the types of *interface*, *library*,

Table 4: Top reused third-party packages

	Import		Clone		Total	
Ethereum Mainnet	@openzeppelin/contracts	29.59%	@openzeppelin/contracts	43.90%	@openzeppelin/contracts	73.46%
	@openzeppelin/contracts-upgradeable	4.25%	@aragon/apps-vault	3.52%	@openzeppelin/contracts-upgradeable	4.50%
	erc721a	2.83%	@pancakeswap/pancake-swap-lib	1.52%	erc721a	3.72%
	hardhat	1.15%	@pooltogether/pooltogether-contracts	0.97%	@aragon/apps-vault	3.52%
	@uniswap/v2-periphery	0.55%	erc721a	0.89%	@pancakeswap/pancake-swap-lib	1.52%
Goerli Testnet	@openzeppelin/contracts	47.97%	@openzeppelin/contracts	0.10%	@openzeppelin/contracts	48.07%
	@openzeppelin/contracts-upgradeable	10.40%	@flair-sdk/contracts	0.04%	@openzeppelin/contracts-upgradeable	10.42%
	hardhat	5.78%	synthetix	0.04%	hardhat	5.78%
	erc721a	3.54%	@thirdweb-dev/contracts	0.04%	erc721a	3.54%
	@uniswap/v3-core	1.05%	@connext/nxtp-contracts	0.03%	@uniswap/v3-core	1.05%

and *abstract* are expected to be reused code blocks. It is hoped that more such reusable code blocks can be extracted or created in the future for safe smart contract development. Furthermore, it is observed that some subcontracts often appear together. For instance, we noticed that IERC721 and IERC165 always come in pairs. This is because IERC721 is an *interface* for standard non-fungible token development, which inherits from IERC165 to make sure compatibility between interfaces. This suggests the potential to discover development patterns for smart contracts.

Finding 6: Subcontracts from well-established third parties are used more frequently than those from unknown sources. The most frequently reused subcontracts are well-defined reusable subcontracts with types of *interface*, *library*, and *abstract*, making up 82% in the top 1,000 reused subcontracts.

Overview of Third-Party Packages. The usage frequency of the third-party packages is summarized in Table 4. The result highlights the wide use of OpenZeppelin [10], the packages for secure smart contract development. On Ethereum Mainnet, over 73% of contracts introduced subcontracts from OpenZeppelin packages. In fact, the top 10 reused third-party subcontracts in Table 2 for both networks are all from OpenZeppelin packages. Apart from that, we identified the frequent use of erc721a, a package that explores gas optimization in batch-minting NFTs for ERC721 tokens. The rest of the frequently reused third-party packages are either for testing (i.e., Hardhat [16]) or token exchange (i.e., Uniswap [19]). Additionally, since a third-party subcontract can be either imported or cloned to a smart contract, we also quantified the use of third-party packages introduced via different approaches, from where we noticed that contracts in Ethereum Mainnet tend to clone subcontracts from OpenZeppelin packages instead of using import statements. Apart from that, the use of some other packages was highlighted. *@aragon/apps-vault* and *@thirdweb-dev/contracts* are for web applications while *@pancakeswap/pancake-swap-lib* is an extension of Uniswap focusing on safety and gas efficiency.

During the examination of third-party packages, the following three types of packages were summarized: ① Packages following EIPs for safe smart contract development (e.g. OpenZeppelin); ② Packages to define or to be used under specific protocols (e.g. Uniswap); ③ Packages for utility and testing (e.g. Hardhat [16]). In our identified third-party packages, 25% of packages follow EIP, the standards suggesting potential new features or processes for Ethereum [8]. The rest packages are either used under specific protocols (65.31%) or for testing and utility (9.69%).

Moreover, to have a better idea regarding the extent of clones in third-party packages, we also performed a subcontract-level clone detection for all identified third-party subcontracts, through which we found 139 subcontracts that appear in more than 10 libraries, 38.85% of which are *interface*. Apart from that, about 90% of the reused subcontracts are sourced from OpenZeppelin. In fact, around 36% of our collected packages are extensions of OpenZeppelin. This also suggests the great influence of OpenZeppelin in smart contract development. While examining the rest 10% reused subcontracts not related to OpenZeppelin, we discovered another group of third-party packages that focus on the decentralized cryptocurrency exchange, which is not covered by OpenZeppelin.

As smart contract development heavily relies on OpenZeppelin packages, it increases the risk of a single point of failure if OpenZeppelin's code or systems were compromised. In fact, we identified an existing dependency management problem regarding OpenZeppelin. While examining OpenZeppelin packages, we noticed an iteration in the name of the packages from *openzeppelin-solidity* to *@openzeppelin/contracts*. Both package names are still in use in current development, though they link to the same package. Moreover, while *@openzeppelin/contracts* is the current official name and has been widely used, the name appearing in the *package-lock.json* file generated by NPM is still *openzeppelin-solidity*. This may result in a miss in vulnerability detections if the vulnerability detection is conducted based on the *package-lock.json* file. Apart from that, the prevalence of OpenZeppelin indicates the need for well-established subcontracts to build secure and reliable smart contracts.

Finding 7: OpenZeppelin [10] has a great influence on third-party subcontracts in smart contract development. Around 90% of the imported third-party subcontracts are sourced from OpenZeppelin, and around 36% of our identified third-party packages are extensions of OpenZeppelin. This could be risky if an OpenZeppelin-related vulnerability is reported.

3.6 Development Patterns (RQ4)

3.6.1 ***Experiments Setup.*** Inspired by the findings of common practices in smart contract development, we further examined the existence of development patterns in our dataset with frequent itemset (FP) mining [31], a well-established technique in data mining. Specifically, each contract is represented by the pre-computed MD5 hashes of its subcontracts, serving as FP mining input to identify subcontract sets that frequently appear together.

To validate the obtained subcontract sets, a manual inspection is conducted with three smart contract auditors. Each auditor first

Table 5: Extracted development patterns

Rank	Pattern Details	Protocols	Context	Problem	Solution	Frequency	Day Range
1	IERC721Metadata, ERC165, IERC721Receiver, IERC721, Strings, Address, Context	EIP721, EIP165	Token creation	Standard token to be owned and traded	Use the ERC721 standard with the Metadata and Receiver interfaces for rich metadata and secure receiving functionality	63,316 (13.72%)	698 days
2	IUniswapV2Router02, IUniswapV2Factory, IERC20, Context, Ownable	EIP20, Uniswap	Token swap	Need to control ownerships	Use Uniswap V2's interfaces with the Ownable contract for permissions	62,855 (21.34%)	699 days
3	IERC721Enumerable, IERC721Metadata, ERC165, IERC721Receiver, IERC721, Strings, Address, Context	EIP721, EIP165	Token creation	Need to enumerate all owned tokens	Use the ERC721 Enumerable extension to allow for token enumeration	53,679 (8.84%)	698 days
4	IUniswapV2Pair, IUniswapV2Router02, IUniswapV2Factory, IERC20, Context, Ownable	EIP20, Uniswap	Token swap	Need to interact with a specific pair of tokens with ownership controls	Use Uniswap V2's Pair and Factory interfaces with the Ownable contract for permissions	14,869 (7.86%)	694 days
5	Address, SafeMath, IERC20, Context, Ownable	EIP20	Token creation	Need for safe math operations and ownership controls	Use the SafeMath library for safe arithmetic operations and the Ownable contract for permissions	14,012 (7.40%)	699 days
6	StorageSlot, Proxy, Address	EIP1967	Proxy contract	Need to upgrade contracts with storing the current state	Use the Proxy pattern along with StorageSlot for preserving the contract's state during upgrades	13,873 (7.33%)	604 days
7	ERC721Creator, StorageSlot, Proxy, Address	EIP721	Proxy contract	Need to upgrade ERC721 token with storing the current state	Use ERC721Creator for creating tokens and the Proxy pattern with StorageSlot for upgradability	12,532 (6.62%)	388 days
8	ReentrancyGuard, IERC721Metadata, ERC165, IERC721Receiver, IERC721, Strings, Address, Context	EIP721, EIP165	Token creation	Prevent reentrancy attack	Use the ReentrancyGuard contract to prevent reentrancy attack	8,579 (4.53%)	684 days
9	MerkleProof, IERC721Metadata, ERC165, IERC721Receiver, IERC721, Address, Context	EIP721, EIP165	Token creation	Need to create and validate proofs	Use the MerkleProof contract to create and validate proofs	4,806 (2.54%)	645 days

independently evaluates if these subcontract sets can be considered as patterns. Disagreements are revisited via group discussion, and auditors can alter their decision once post-discussion. We then collate the results, identifying development patterns following the rule that the majority wins. We further validate the patterns with frequency and date range. Frequency quantifies the appearance of a pattern while date range represents the time span from the first to the last appearance of a pattern. These provide insights into the prevalence and longevity of patterns in contract development.

3.6.2 ***Results and Discussion.*** We finally identified 61 development patterns, 6 of which appeared over 10,000 times and an additional 13 appeared more than 1,000 times in our dataset. The top 9 development patterns are presented in Table 5 while the full list of identified patterns is available on our website [17].

Among the identified patterns, over 63% are related to standard token creation, specifically, 74% involving ERC721 tokens. For example, the 3rd pattern is to create enumerable tokens, the 8th is to prevent reentrancy attacks [37], one of the most common attacks for smart contracts, and the 9th is to create and validate proofs. Other patterns are related to proxy contracts and token swaps, accounting for 20.31% and 10.94% respectively. The appearance of patterns suggests that the creation of tokens, token swaps, and proxy contracts have been highly standardized, reflecting the common functional requirements and architectural paradigm in development.

Examples of contract addresses for each pattern can be found on our website [17] to help understand the application of patterns in real-world development. For example, the second pattern is to facilitate the ERC20 token swap via the Uniswap V2 protocol [19]. To utilize this pattern, an additional customized subcontract is often required to define token details. An example is shown in Listing 1, in which the token name, symbol, and other relevant information are defined. In fact, this can be handled by low-code development easily. Since this customized subcontract usually follows a similar pattern, user-extracted information can be integrated into a template and merged together with the pre-defined pattern.

Additionally, we also conducted the same analysis in smart contracts from Goerli Testnet. Apart from the already identified development patterns in Ethereum Mainnet, Goerli Testnet contains development patterns sourced from online tutorials. For example, [21] is a template for simple storage provided by an online course, which appears over 800 times in our dataset. While using patterns can improve code quality and mitigate potential vulnerabilities, it is crucial to ensure the safety of the patterns.

```
pragma solidity ^0.8.0;
contract ThePathless is Context, IERC20, Ownable {
    using SafeMath for uint256;
    string private constant _name = "The Pathless";
    string private constant _symbol = "Pathless";
    ......
    constructor() {
        _rOwned[_msgSender()] = _rTotal;
        IUniswapV2Router02 _uniswapV2Router =
             IUniswapV2Router02(0
             x7a250d5630B4cF539739dF2C5dAcb4c659F2488D);
        uniswapV2Router = _uniswapV2Router;
        uniswapV2Pair = IUniswapV2Factory(_uniswapV2Router.
             factory()).createPair(address(this),
             _uniswapV2Router.WETH());
        _isExcludedFromFee[owner()] = true;
        _isExcludedFromFee[address(this)] = true;
        _isExcludedFromFee[_developmentAddress] = true;
        _isExcludedFromFee[_marketingAddress] = true;
        emit Transfer(address(0), _msgSender(), _tTotal);
    } // main body
}
```

Listing 1: An application example for Rank 2 pattern in Table 5

During our inspections, the patterns we identified are generally secured since most subcontracts are from audited third-party packages such as Openzeppelin [10] and Uniswap [19]. In addition, most identified patterns follow EIPs, which helps to enhance their security, interoperability, and compliance with community standards. However, the requirement of customization for many of the patterns could potentially introduce risks if not properly handled.

Finding 8: We identified 61 development patterns. Specifically, token creation, proxy contracts, and token swaps make up 68.75%, 20.31%, and 10.34% of the identified patterns, signifying high standardization in these areas.

4 IMPLICATIONS AND LESSONS

4.1 Result Summary

Unlike existing works ignoring the common practices of reusing external subcontracts, in this study, we investigate code reuse in smart contracts with considerations of dependency management. First,

we statistically confirmed the high ratio of reusing external subcontracts (82.94%), aligning with previous works [33, 35]. However, after investigating their introduction approaches, we notice that 31.22% of subcontracts were introduced by *imports* instead of simple code clones. Next, we divided subcontracts into self-developed (17.06%) and external ones for further code reuse analysis. For instance, we observe the reliance on function reuse increases when the number of functions in self-developed subcontracts grows. In addition, over 35% of the external subcontracts are *interface*, indicating a trend toward adhering to established standards and protocols. Yet, subcontract types are sometimes misused, posing challenges in code management. Moreover, OpenZeppelin influences around 90% of the imported external subcontracts, highlighting its prominent role in contract development. Lastly, we identified 61 development patterns for token creation (68.75%), proxy contracts (20.31%), and token swaps (10.34%), reflecting areas with high standardization.

4.2 Lessons Learned

In this section, we summarize the discovered challenges in smart contract development and proposed possible solutions.

For Contract Developers. ① Though various approaches are available to introduce external subcontracts as discussed in Section 3.3, we suggest avoiding using URLs to import external subcontracts unless the URL has been thoroughly verified and audited for security development. This is to avoid introducing untrustworthy code blocks into contracts due to malicious URLs or compromised subcontracts. ② As discussed in Section 3.3, over 80% subcontracts are introduced externally in smart contracts. However, the documentation of such usage often misses, especially after on chains. This brings challenges in the management of external subcontracts and more critically, in the identification and mitigation of vulnerabilities in external subcontracts. Hence, we would recommend including basic information about each subcontract such as their origins and versions if available in the published source code. This practice would help with code maintenance and management of dependency downstream. ③ The inconsistent use of subcontract types pointed out in Section 3.5 may further complicate code organization and management instead, indicating a need for enhanced standardization in contract development.

For Third-party Auditors. ① Considering the high code reuse ratio (82.94%) highlighted in Section 3.3, it may help to boost the efficiency of audits by establishing a standardized knowledge-based vulnerability database that can be accessed and contributed by every auditor, especially for frequently reused components. Once the database is established, the relevant information for external components can be identified by code clone detection, avoiding repeated audits on the same components. ② Given the heavy reliance on external subcontracts, especially those from well-established third parties as mentioned in Section 3.5, it is important to regularly audit the packages to ensure their security and reliability. Moreover, third-party packages are not used as a whole in contract development, instead, developers only import required subcontracts. Hence, it is important to link identified vulnerabilities to specific subcontracts or functions instead of packages. ③ Though there are auditions done with the published packages with specific smart contract applications, we do not see any auditions with the frequently used development patterns as identified in Section 3.6. However, it is important to continuously monitor and manage such patterns to make sure safety, maintainability, and reliability.

For the Community. While most third-party packages are managed under NPM as detailed in Section 3.3, it would be good to have a specialized package manager for smart contract development to better manage the dependencies of smart contracts during development and after deployment. As a specialized package manager for Solidity smart contracts, it would be good to have: ① Automated security analysis. The package manager should continuously access the security and use of the third-party packages. ② Package versioning. The package manager should have a robust versioning system to keep track of the updates and changes in smart contract packages. ③ Easy integration. The package manager should be easy to integrate with popular development tools and environments to make smart contract development smoother. ④ Package deployment. The package manager is expected to continuously monitor and manage the used third-party subcontracts even after deployment. Additionally, considering the dominance of OpenZeppelin packages identified in Section 3.5, it would be beneficial to encourage and support the development of alternative and secure packages to provide more options for smart contract developers.

4.3 Future Research Directions

Our findings suggest several directions for future research. First, the prevalent code reuse in smart contracts necessitates studying vulnerability spread across chains and the efficiency of countermeasures. There is potential in developing detectors for clone-based vulnerabilities to auto-patch risky and employing binary code clone detection to improve vulnerability identification. Second, extending our analysis to include type-3 clones could provide deeper insights into diverse implementations of identical functionalities. The usage scenarios for various implementations could be further studied. In addition, our analysis can be extended by including multiple blockchain networks (e.g., BNB Chain [24] and Polygon [22]). This will enhance our understanding of contract characteristics and cross-chain interactions, enriching our grasp on the broader smart contract ecosystem. Lastly, with the development patterns provided in Section 3.6, their effects on contract robustness and risks and the evolution of the patterns could be further investigated. Moreover, investigating functional-level patterns could offer insights into subcontract versions and boost low-code development flexibility.

4.4 Threats to Validity

4.4.1 ***Internal Validity.*** The choice of type-2 clones may lead to false similarities in clone detection. In this study, we chose to focus on type-2 clones because they offer a balance between the rigor of detection and the richness of insight. For example, our duplicated contract analysis in Section 3.3 can be effectively done with type-2 clones, but unattainable with type-1 clones. However, we also acknowledge the risk of identifying false similarities with type-2 clones due to the renaming of variable names. To mitigate this, we have reconducted experiments with type-1 clone detection as a baseline and the result proves that our initial findings still hold.

4.4.2 ***External Validity.*** A potential threat is related to the representativeness of the dataset. Collecting the source code of all

deployed contracts is challenging due to the frequent lack of public release and the numerous existing blockchain networks. To mitigate this, we expand our dataset by collecting contracts from two different Ethereum networks (i.e., the Mainnet and Goerli Testnet) and identified third-party packages.

4.4.3 ***Construct Validity.*** The identified patterns in Section 3.6 might be influenced by our subjective interpretations. We mitigate this with a rigorous inspection process involving three smart contract auditors and group discussions for cross-verification. We then revalidate the results using frequency and date range to measure the prevalence and longevity of identified patterns. Through these verifications, we ensure that these development patterns are prevalent and can be applied in real-world applications. Similar inspections are also conducted during duplicated contract analysis in Section 3.3. The result is further verified with the information of creators from source code and Etherscan [7]. The records for both manual inspections are available on our website [17].

5 RELATED WORK

Empirical Studies on Smart Contracts. Several empirical studies on code reuse in smart contracts have been done in the past few years with focuses on the characteristics [33, 35] and impacts [28, 41] of clones. Kondo et al. [35] reported that 79.2% of the subcontracts are type-1 and type-2 clones in Ethereum with a tree-based clone detector called Deckard [32]. While Khan [33] further extended Kondo's study to the granularity of functions and type-3 clones. Their result indicated a 30.13% overall clone ratio at the function level, out of which about 90% belong to type-1 clones. Chen et al. [28] studied the impacts of code reuse in smart contracts and identified the common revisions to reuse external subcontracts. Pierro et al. [41] classified the clones in smart contract development and proposed some possible explanations for trends of clones in the view of smart contract developers. While previous studies concluded a high clone ratio in smart contracts, they overlooked that many external subcontracts are directly imported but published together with the source code. In contrast, we take the introduction approaches into consideration during analysis. This allows us to better comprehend the existing regulations to use external components and uncover that over 20% of the clones are imported.

Additionally, there are a few studies discussing the current smart contract development from various aspects. Zou et al. [51] summarized the major challenges for developers in smart contract development and discussed the possible directions to improve the quality of current smart contract development. Chen et al. [27] studied the maintenance-related concerns for deployed smart contracts and discussed how to maintain smart contract-based projects from developers' views. Several studies have been to discuss the concerns in smart contract development such as security [26, 43, 45], and maintenance [44]. However, these studies only focused on a single granularity (i.e., either subcontracts or functions), thereby leaving the composition of smart contracts under-explored. Our study bridges this gap by providing a detailed understanding of smart contract composition and conducting code reuse for multiple granularities (i.e. contract, subcontract, and function). For each level of granularity, we manage to discover some common practices in code reuse and functional requirements.

Development Patterns in Smart Contract Development. A few studies related to the existing development patterns have been done in recent years. Marino and Juels [38] proposed design patterns for altering and undoing smart contracts. Bartoletti and Pompianu [25] manually checked the source code of 811 verified smart contracts from Etherescan, through which they summarized the purposes of smart contracts (e.g. financial, wallet, etc.) and existing design patterns including token, authorization, and oracle. Similar design patterns were also discussed in Worley's work [47]. Oliva et al. [40] investigated the activity level of contracts and summarized the main usage of smart contracts. Wohrer et al. [46] elaborated several common security patterns and described solutions to typical attack scenarios. Zheng et al. [49] summarized the whole life cycle of smart contract development, including contract creation, deployment, execution, and completion. Moreover, they also discussed the challenges in the current development cycle and the difference between smart contract platforms including Ethereum. Another work from Zheng [50] described the architecture of blockchain and consensus algorithms and introduced the development of applications based on blockchain. Mohanta et al. [39] summarized seven use cases for smart contracts, including IoT, supply chain management, and healthcare systems. Despite these contributions, previous studies failed to provide concrete development patterns but focused on the general steps for various functional requirements. In our study, we extract development patterns as sets of subcontracts that could be potentially used for low-code developments. This also reflects the areas with high standardization in smart contract development.

6 CONCLUSION

This study discusses code reuse in Solidity smart contracts from the view of contract decomposition on Ethereum blockchains. To understand the compositions of contracts, we first quantified three common approaches to introduce subcontracts, through which we also compiled a list of commonly used third-party packages in smart contract development. We then conducted separate code reuse analyses for self-developed and external subcontracts to dissect the most frequently reused code blocks and their functional properties. To gain insights into common practices in contract development, we also extracted 61 frequently reused development patterns. Lastly, we discussed the current challenges in managing smart contracts and proposed possible solutions.

7 DATA AVAILABILITY

The detailed analysis results that support the findings of this study are available in [17].

ACKNOWLEDGEMENT

This research is supported by the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore and Cyber Security Agency of Singapore.

REFERENCES

[1] 2022. https://github.com/ethereum/solidity.
[2] 2022. https://github.com/.
[3] 2022. https://docs.ipfs.tech/concepts/ipfs-gateway/.
[4] 2022. https://gateway.ethswarm.org/.
[5] 2022. https://remix-ide.readthedocs.io/en/latest/import.html.
[6] 2022. ANTLR v4. https://github.com/antlr/antlr4 original-date: 2010-02-04T01:36:28Z.
[7] 2022. Ethereum (ETH) Blockchain Explorer. http://etherscan.io/.
[8] 2022. Ethereum Improvement Proposals (EIPs). https://github.com/ethereum/EIPs original-date: 2015-10-26T13:57:23Z.
[9] 2022. npm. https://www.npmjs.com/
[10] 2022. OpenZeppelin/openzeppelin-contracts. https://github.com/OpenZeppelin/openzeppelin-contracts original-date: 2016-08-01T20:54:54Z.
[11] 2023. anyswap/multichain-smart-contracts: multichain smart contracts. https://github.com/anyswap/multichain-smart-contracts. (Accessed on 01/23/2023).
[12] 2023. Contracts — Solidity 0.8.17 documentation. https://docs.soliditylang.org/en/v0.8.17/contracts.html. (Accessed on 01/29/2023).
[13] 2023. EIPs/eip-20.md at master · ethereum/EIPs. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md. (Accessed on 01/31/2023).
[14] 2023. EXPLAINED: THE MULTICHAIN HACK (JANUARY 2022). https://halborn.com/explained-the-multichain-hack-january-2022/. (Accessed on 01/23/2023).
[15] 2023. goerli.etherscan.io. https://goerli.etherscan.io/. (Accessed on 01/23/2023).
[16] 2023. Hardhat | Ethereum development environment for professionals by Nomic Foundation. https://hardhat.org/. (Accessed on 01/27/2023).
[17] 2023. Home. https://sites.google.com/view/solidity-contract-analysis/home. (Accessed on 02/02/2023).
[18] 2023. Home | ethereum.org. https://ethereum.org/en/. (Accessed on 01/20/2023).
[19] 2023. Home | Uniswap Protocol. https://uniswap.org/. (Accessed on 01/27/2023).
[20] 2023. manifoldxyz/creator-core-solidity. https://github.com/manifoldxyz/creator-core-solidity. (Accessed on 06/11/2023).
[21] 2023. PatrickAlphaC/storage-factory-fcc. https://github.com/PatrickAlphaC/storage-factory-fcc. (Accessed on 02/01/2023).
[22] 2023. polygon.technology. https://polygon.technology/. (Accessed on 06/25/2023).
[23] 2023. Smart contract - Wikipedia. https://en.wikipedia.org/wiki/Smart_contract. (Accessed on 01/24/2023).
[24] 2023. www.bnbchain.org. https://www.bnbchain.org/. (Accessed on 06/25/2023).
[25] Massimo Bartoletti and Livio Pompianu. 2017. An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns. *Lecture Notes in Computer Science* (03 2017). https://doi.org/10.1007/978-3-319-70278-0_31
[26] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 454–469.
[27] Jiachi Chen, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. 2021. Maintenance-related concerns for post-deployed Ethereum smart contract development: issues, techniques, and future challenges. *Empirical Software Engineering* 26, 6 (2021), 1–44.
[28] Xiangping Chen, Peiyong Liao, Yixin Zhang, Yuan Huang, and Zibin Zheng. 2021. Understanding Code Reuse in Smart Contracts. 470–479. https://doi.org/10.1109/SANER50967.2021.00050
[29] W. G. Cochran. 1934. The distribution of quadratic forms in a normal system, with applications to the analysis of covariance. *Mathematical Proceedings of the Cambridge Philosophical Society* 30, 2 (1934), 178–191. https://doi.org/10.1017/S0305004100016595
[30] Token Generator. 2023. Token Generator | Create ERC20 or BEP20 Token | SmartContracts Tools. https://www.smartcontracts.tools/token-generator/. (Accessed on 02/03/2023).
[31] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. 2007. Frequent pattern mining: current status and future directions. *Data mining and knowledge discovery* 15, 1 (2007), 55–86.
[32] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 96–105.
[33] Faizan Khan, Istvan David, Daniel Varro, and Shane McIntosh. 2022. Code Cloning in Smart Contracts on the Ethereum Platform: An Extended Replication Study. *IEEE Transactions on Software Engineering* (2022), 1–13. https://doi.org/10.1109/TSE.2022.3207428 Conference Name: IEEE Transactions on Software Engineering.
[34] Shafaq Naheed Khan, Faiza Loukil, Chirine Ghedira-Guegan, Elhadj Benkhelifa, and Anoud Bani-Hani. 2021. Blockchain smart contracts: Applications, challenges, and future trends. *Peer-to-peer Networking and Applications* 14, 5 (2021), 2901–2925.
[35] Masanari Kondo, Gustavo A. Oliva, Zhen Ming (Jack) Jiang, Ahmed E. Hassan, and Osamu Mizuno. 2020. Code cloning in smart contracts: a case study on verified contracts from the Ethereum blockchain platform. *Empirical Software Engineering* 25, 6 (Nov. 2020), 4617–4675. https://doi.org/10.1007/s10664-020-09852-5
[36] Paul Levy. 2014. *Finite Population Correction*. https://doi.org/10.1002/9781118445112.stat05700
[37] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. ReGuard: Finding Reentrancy Bugs in Smart Contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 65–68. https://doi.org/10.1145/3183440.3183495
[38] Bill Marino and Ari Juels. 2016. Setting Standards for Altering and Undoing Smart Contracts, Vol. 9718. 151–166. https://doi.org/10.1007/978-3-319-42019-6_10
[39] Bhabendu Kumar Mohanta, Soumyashree S Panda, and Debasish Jena. 2018. An Overview of Smart Contract and Use Cases in Blockchain Technology. In *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. 1–4. https://doi.org/10.1109/ICCCNT.2018.8494045
[40] Gustavo A. Oliva, Ahmed E. Hassan, and Zhen Ming (Jack) Jiang. 2020. An exploratory study of smart contracts in the Ethereum blockchain platform. *Empirical Software Engineering* 25, 3 (May 2020), 1864–1904. https://doi.org/10.1007/s10664-019-09796-5 Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 3 Publisher: Springer US.
[41] Giuseppe Antonio Pierro and Roberto Tonelli. 2021. Analysis of Source Code Duplication in Ethreum Smart Contracts. 701–707. https://doi.org/10.1109/SANER50967.2021.00089
[42] Chanchal Roy and James Cordy. 2007. A Survey on Software Clone Detection Research. *School of Computing TR 2007-541* (Jan. 2007).
[43] Amritraj Singh, Reza Meimandi Parizi, Qi Zhang, Kim-Kwang Raymond Choo, and Ali Dehghantanha. 2020. Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. *Comput. Secur.* 88 (2020).
[44] 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.
[45] Zhiyuan Wan, Xin Xia, David Lo, Jiachi Chen, Xiapu Luo, and Xiaohu Yang. 2021. Smart contract security: A practitioners' perspective. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1410–1422.
[46] 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, 2–8.
[47] Carl R. Worley and Anthony Skjellum. 2018. Opportunities, Challenges, and Future Extensions for Smart-Contract Design Patterns. In *Business Information Systems*.
[48] Jiahui Wu, Zhengzi Xu, Wei Tang, Lyuye Zhang, Yueming Wu, Chengyue Liu, Kairan Sun, Lida Zhao, and Yang Liu. 2023. OSSFP: Precise and Scalable C/C++ Third-Party Library Detection using Fingerprinting Functions. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 270–282. https://doi.org/10.1109/ICSE48619.2023.00034
[49] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. 2020. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems* 105 (2020), 475–491.
[50] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. 2018. Blockchain challenges and opportunities: A survey. *International Journal of Web and Grid Services* 14 (10 2018), 352. https://doi.org/10.1504/IJWGS.2018.095647
[51] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2019. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering* 47, 10 (2019), 2084–2106.

Received 2023-02-02; accepted 2023-07-27