Revealing Hidden Threats: An Empirical Study of Library Misuse in Smart Contracts

Mingyuan Huang
Sun Yat-Sen University
China
huangmy83@mail2.sysu.edu.cn

Jiachi Chen
Sun Yat-Sen University
China
chenjch86@mail.sysu.edu.cn

Zigui Jiang
Sun Yat-Sen University
China
jiangzg3@mail.sysu.edu.cn

Zibin Zheng*
Sun Yat-Sen University
China
zhzibin@mail.sysu.edu.cn

ABSTRACT

Smart contracts are Turing-complete programs that execute on the blockchain. Developers can implement complex contracts, such as auctions and lending, on Ethereum using the Solidity programming language. As an object-oriented language, Solidity provides libraries within its syntax to facilitate code reusability and reduce development complexity. Library misuse refers to the incorrect writing or usage of libraries, resulting in unexpected results, such as introducing vulnerabilities during library development or incorporating an unsafe library during contract development. Library misuse could lead to contract defects that cause financial losses. Currently, there is a lack of research on library misuse. To fill this gap, we collected more than 500 audit reports from the official websites of five audit companies and 223,336 real-world smart contracts from Etherscan to measure library popularity and library misuse. Then, we defined eight general patterns for library misuse; three of them occurring during library development and five during library utilization, which covers the entire library lifecycle. To validate the practicality of these patterns, we manually analyzed 1,018 real-world smart contracts and publicized our dataset. We identified 905 misuse cases across 456 contracts, indicating that library misuse is a widespread issue. Three patterns of misuse are found in more than 50 contracts, primarily due to developers lacking security awareness or underestimating negative impacts. Additionally, our research revealed that vulnerable libraries on Ethereum continue to be employed even after they have been deprecated or patched. Our findings can assist contract developers in preventing library misuse and ensuring the safe use of libraries.

KEYWORDS

Blockchain, Ethereum, Library Misuse, Empirical Study

*corresponding author

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3623335

ACM Reference Format:
Mingyuan Huang, Jiachi Chen, Zigui Jiang, and Zibin Zheng. 2024. Revealing Hidden Threats: An Empirical Study of Library Misuse in Smart Contracts. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3623335

1 INTRODUCTION

The success of blockchain technology has been attracting great attention from both industry and academia. Being the first public blockchain platform to support smart contracts, Ethereum has become one of the most popular blockchain platforms, reaching a global market cap of $215 billion by March 2023 [38]. Smart contracts on Ethereum are Turing-complete programs that can be developed through the Solidity programming language. Developers can implement complex business scenarios with smart contracts such as auctions, lending, and crowdfunding. The complexity of smart contract codes has also increased with the growing user demands. By December 2022, the average line of codes in Ethereum contracts has reached 860 lines.

Similar to other Object-oriented programming languages, Solidity provides various libraries [17] which are collections of reusable functions that can be utilized by other contracts. These libraries allow developers to create more efficient, modular, and scalable contracts by separating common functionality into separate, shareable entities. By December 2022, 156,761 (70%) of contracts utilized libraries, demonstrating their widespread use.

The lifecycle of a library typically involves three key roles: the library developer who creates the library, the contract developer who utilizes the library, and the community who is responsible for maintaining the library. There are several communities that maintain some wildly-used libraries and play a crucial role in the Ethereum ecosystem. For example, Openzeppelin contracts [19] is an online database recommended by the Solidity official document [18]. this repository has maintained commonly used Solidity libraries for secure smart contract development since July 2017. By December 2022, Openzeppelin has amassed 28 libraries, with the top 10 popular libraries utilized 289,927 times.

Effective collaboration among these three roles can facilitate smart contract development. However, inappropriate behaviors by either library developers or contact developers can result in unexpected negative consequences. We refer to these library-related improper behaviors as library misuse. For library developers, it

is essential to prioritize the security of libraries to prevent vulnerabilities, as the code can be reused in multiple contracts. In Ethereum, 75% of libraries are utilized across multiple contracts, and any vulnerabilities presented in these libraries will have a significant negative impact due to their widespread use. For contract developers, inappropriate utilization of a library may cause the library to be unable to perform its intended features and even result in financial losses. For example, BeautyChain citebeauty is the victim of an integer overflow attack. Although BeautyChain introduced the SafeMath, which contains wrappers over arithmetic operations to defend against overflow attacks, the improper library usage still led to the contract owner losing over one billion dollars. The term "library misuse" collectively refers to the issues that occur in both library development and library utilization. This includes engaging in incorrect behaviors throughout the lifecycle of the library. Therefore, it is crucial to avoid library misuse in both the development and utilization of libraries.

In this paper, we conduct an empirical study on smart contract library misuse. First, we collect 223,336 smart contracts from Ethereum by December 2022 and analyze the popularity of 3,864 real-world libraries in total. We found that the top 100 popular libraries (2.6%) are utilized 351,926 (90%) times, and analyzing popular libraries has a higher priority due to their wide impact. Then, we extracted and categorized 105 library misuse instances based on 593 audit reports published by five audit companies, using the open card sorting approach [65]. We identify eight misuse patterns from the 105 instances. We illustrate the causes and impacts of each pattern, and give corresponding code examples which can help researchers and developers understand them.

To verify the practicality of our patterns, we analyze 1018 real-world contracts and found 905 misuse cases in 456 contracts, which demonstrates widespread misuse in the real world. The frequency of different patterns is highly varied, with 99% of all identified cases concentrated in three subtle patterns. The most common pattern occurred 543 times in 204 contracts, with 83 of those contracts containing the same type of misuse in different code locations. This suggests a lack of comprehension of certain misuse patterns by developers in the real world. In addition, we also analyze the propagation of vulnerable libraries in real-world contracts from 2015 to 2022. We find that some vulnerable libraries on Ethereum are still utilized despite being deprecated or fixed by the maintainer, and some libraries continue to propagate for three years due to developers obtaining libraries from unreliable sources instead of reputable communities (e.g., Openzeppelin).

The main contributions of this paper are as follows:

- We propose eight patterns of library misuse to help developers prevent library misuse and potential financial losses. To the best of our knowledge, this is the first empirical study on library misuse.
- We analyze real-world library misuse; researchers and developers can accordingly understand the proportion and impact of different misuse patterns on Ethereum and allocate reasonable efforts to promote better practice.
- We provide the first dataset [42] of library misuse, including 92 misuse cases from audited contracts and 905 misuse cases from real-world contracts.

The subsequent sections of this paper are structured as follows. In Section 2, we provide the background knowledge of smart contracts and Solidity libraries. Then, we introduce the investigation progress for collecting and categorizing library misuse cases in Section 3. Section 4 outlines eight specific library misuse patterns with examples. We then evaluate patterns on real-world contracts in Section 5 and address potential validity threats in Section 6. In Section 7, we offer suggestions for developing detection tools for library misuse. Related works are introduced in Section 8. Finally, we conclude the entire research in Section 9.

2 BACKGROUND

2.1 Ethereum and Smart Contracts

Ethereum [14] is the first blockchain platform that supports smart contracts. Ethereum has two types of accounts: external owned accounts (EOAs) and contract accounts (CAs). EOAs are controlled by users who hold the corresponding private keys, while CAs are controlled by smart contracts. A smart contract [14] is a Turing-complete program running on the blockchain. Smart contracts can be implemented in various scenarios (e.g., auctions, exchanges, crowdfunding, etc.) using high-level programming languages such as Solidity [40]). These smart contracts are often wrapped into user-friendly Apps, hence named Decentralized Apps (DApps).

2.2 Audit Reports

Smart contracts are often used to manage digital assets, making them attractive targets for hacking attacks. To mitigate these risks, some security companies (e.g., PeckShield [34]) offer auditing services for these contracts. These services typically involve manual analysis by security experts who identify and assess potential vulnerabilities in the contract and issue audit reports. Many smart contracts published their audit reports, thereby providing users with evidence of contract security. Out of the 593 audit reports we have gathered, each report has 2,956 words (17 pages) on average. These audit reports provide a detailed description, code location, and recommendation for fixing each vulnerability. They are typically presented in English to ensure convenience for global investors.

2.3 Solidity Library

A Solidity library [14] comprises a set of reusable functions contract developers can utilize to optimize tasks. Libraries are usually designed to solve specific common problems and facilitate code reuse. Code reuse [53, 59] is a software engineering concept that suggests building new programs using pre-existing code. The Library-based approach has been adopted by many programming languages (e.g., C++ [63], Java[54], etc.). In Solidity, a library is defined by the "library" keyword and linked to the contract during the compilation. We list below a few popular libraries.

SafeMath: SafeMath [29] is a library that provides a secure and reliable way to perform arithmetic operations in Solidity-based smart contracts. This library includes functions for addition, subtraction, multiplication, and division, which perform checks to ensure that the result does not exceed the maximum or minimum value of the data type used. Using SafeMath, developers can ensure that arithmetic operations are secure and prevent their contracts from overflow attacks.

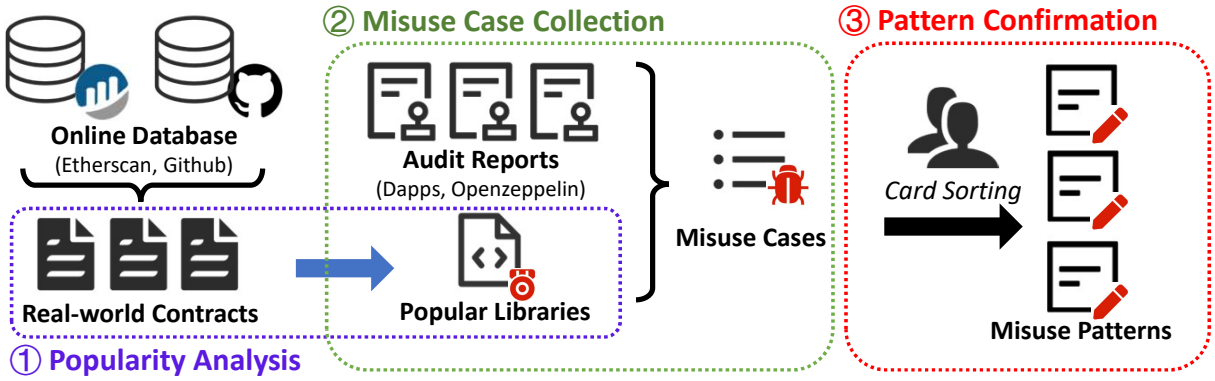

Figure 1: Investigation process for collecting and categorizing library misuse cases

Address: Address [21] library provides wrapper functions for secure interactions with Ethereum addresses. It enables developers to perform common address-related operations, such as transferring Ether, identifying address types. There are four major functions in this library. Function *functionCall*, *functionDelegateCall*, *functionStaticCall* are wrappers over low level *call*, *delegatecall* and *staticcall*. The function *isContract* checks the length of smart contract code via *extcodesize*, and returns true when the length is not zero. The wrapper methods of this library perform the necessary return value checking and exception handling to ensure proper interaction with other addresses, which can prevent unexpected errors during address interaction.

SafeERC20: SafeERC20 [28] is a library designed for safely operating ERC20 tokens [43]. This library helps to prevent common errors, such as tokens lost due to incorrect approvals. There are five major functions in this library, including *safeTransfer*, *safeTransferFrom*, *safeApprove*,*safeIncreaseAllowance* and *safeDecreaseAllowance*, which perform additional checks before executing token transfer or approval. These checks include verifying the transfer or approval was successful, the target address is a valid address, and the sender has sufficient token balance or allowance. Function *safeApprove* only sets allowance when the allowance value is zero, and function *safeIncreaseAllowance* and *safeDecreaseAllowance* are use to ensure a safe allowance change. SafeERC20 is a valuable tool for ensuring the security and integrity of ERC20 token operations.

ECDSA: ECDSA [23] library can verify a signature based on Elliptic Curve Digital Signature Algorithm. There are two core functions in the library, function *recover* accepts the signature and the hashed message as inputs and returns the address that signed the hashed message. This function divides the signature into three variables: r, s, v. Due to the mathematical properties of the elliptic curve, two public keys will be calculated via r and s, while v can be utilized to identify which key was used to produce the signature. And *toEthSignedMessageHash* function can transfer a message hash to an "eth-signed" message that contains an extra prefix.

3 LIBRARY MISUSE INVESTIGATION

Smart contract vulnerabilities have been put into a large amount of effort, but there is no research about library misuse. To address this gap, we conduct an investigation to analyze library misuses.

3.1 Overview

In this study, we focus on the wildly-used libraries in real-world applications and the associated cases of misuse. As shown in Figure 1, we designed a three-step process to discover library misuse cases.

Firstly, we collect real-world Ethereum smart contracts from online databases and analyze the popularity of library usage in these contracts. This analysis helps us gain insight into the actual prevalence of libraries in real-world scenarios. Secondly, we gather audit reports that examine the security issues of DApp contracts and extract all misuse cases related to popular libraries. Finally, we employ the card sorting method to manually identify eight misuse patterns from a total of 92 misuse cases.

3.2 Library Popularity Analysis

To analyze the usage of libraries in real-world Ethereum smart contracts, we initially crawl the source code of the smart contracts. Etherscan [38] provides an interface to query contracts based on their addresses. However, it does not provide any interface that allows the retrieval of all contract addresses; it only provides access to the last 500 contract addresses. To overcome this limitation, we utilize Smart Contract Sanctuary [35], a GitHub project that continuously collects contract addresses provided by Etherscan. Based on the address list accumulated by Smart Contract Sanctuary, we crawl a total of 223,336 Ethereum smart contracts from Etherscan, encompassing data up to December 2022.

To extract library information, we utilize Abstract Syntax Tree (AST) parsing [45, 51], which involves analyzing the structure of the contract's source code. By utilizing ASTs, we extract the names of each contract's libraries, their frequency of usage, and the corresponding code blocks of the libraries. Based on the collected data, we found 3,864 types of libraries used in 156,761 smart contracts 391,073 times. This indicates that more than 70% of the smart contracts employ libraries, and some contracts utilize multiple libraries. Hence, libraries are extensively utilized in the smart contracts.

Then, we rank all 3,864 libraries based on their usage frequency across 391,073 cases and observe that some libraries are frequently used. Specifically, the 100 most popular libraries (2.6%) are utilized in 351,926 library usage cases (90%), and the top 10 widely-used libraries (0.25%) account for 289,927 cases (74%). In contrast, 2,385 libraries (62%) were only employed in fewer than ten cases, and

1,004 libraries (25%) were exclusively adopted in a single contract. As a result, popular libraries demand greater attention due to their significant influence, while a substantial portion of unpopular libraries are of lesser importance. We list the top 10 most popular libraries and their features in Table 1. Four libraries are wrappers to ensure operational security, four libraries are toolkits for convenient development, and two libraries (Counters, EnumerableSet) support new data structures.

3.3 Misuse Case Collection

To collect instances of library misuse, we first need to find corresponding textual descriptions from open-source documents. We choose audit reports as the data source for our investigation. These reports are considered reliable since security experts manually review them. Additionally, audit reports offer detailed descriptions of issues, including vulnerability descriptions and code examples.

Based on the recommendations from the Ethereum official document [5] and Etherscan [36], we obtained the website addresses of these reputable audit companies. Firstly, we excluded companies that do not directly open source audit reports (e.g., Blockchain Consilium [6] only provides query interfaces to contract owners). Subsequently, we filtered out audit companies that had a very small number (< 10) of published reports (e.g., CertK [7] had only 4 reports at the time of our analysis in December 2022). Additionally, we eliminated companies that do not mention any issues related to the library through keyword searches. For example, SmartDec [8], whose available audit reports do not mention any library misuse. Finally, we collected 593 DApp audit reports from 5 audit companies (BlockSec, HAECHI, PeckShield, SlowMist, Solidified). Out of the 593 audit reports we have gathered, each report has 2,956 words (17 pages) on average. These audit reports provide a detailed description, code location, and recommendations for fixing each vulnerability. They are typically presented in English to ensure convenience for global investors.

However, only investigating DApp audit reports will miss some misuse situations, as many libraries are consistently maintained and optimized by developers. It is challenging to cover all versions of libraries, as one audit report usually only analyzes a single version of the library. For example, Openzeppelin contracts have been maintained since they were released on Jul. 2017. The GitHub repository of openzeppelin contracts provides two audit reports for v1.0 and v2.0, but the code version has been updated to v4.0. Some issues after v2.0 have not been summarized in audit reports, but they are already mentioned on the Security Advisor [33] page in the Github repository, which lists recent vulnerabilities by maintainers. Therefore, we also analyze the Security Advisories page as an audit report to help us patch the vulnerabilities in the latest library versions.

We utilize the keyword search method to identify the descriptions of library misuse from audit reports. We select 2,860 library names as keywords in all 3,864 libraries, and filter out the 1004 least popular libraries, as they are never reused in other Ethereum contracts. Then, we manually remove invalid mentions (such as comments about libraries in the code), and eventually identified 92 text descriptions of library misuse.

3.4 Pattern Confirmation

We follow card sorting [65] to categorize all 92 cases related to library misuse. Card sorting is a widely used method for classifying data into different categories. There are three types of card sorting to choose from: closed card sorting, open card sorting, and hybrid card sorting. Closed card sorting requires participants to cluster data into predefined categories, while open card sorting does not provide categories, and participants must define them during the sorting process. Hybrid card sorting combines the two methods, allowing the addition of new categories to predefined ones. Since no previous work has analyzed library misuse patterns, we opt for open card sorting to uncover the categories.

The open card sorting process involves three individuals, including one of the paper authors and two volunteers with more than two years of experience in Solidity development. The author extracts the necessary information for categorization from the corresponding audit report for each misuse case and converts it into a card. For each vulnerability related to library misuse, the author completely reproduces the description of the vulnerability and code examples to a card, and also adds the description location in the audit report as the reference. During card sorting, two volunteers are asked to independently sort the cards into groups with similar characteristics and name each category according to the misuse reason. Then, the authors organize a meeting between these two volunteers. With the confirmation of the two volunteers, the authors unify the category names and ensure each one is based on the same rationale.

Table 1: Top 10 popular libraries used in Ethereum contracts.

Library	Description
SafeMath [29]	Wrappers over arithmetic operations.
Address [21]	Wrappers over address operations.
Strings [31]	Toolkit of string type.
SafeERC20 [28]	Wrappers over ERC20 operations.
StorageSlot [30]	Toolkit of storage slots.
Counters [22]	Supporting Counter in Solidity.
EnumerableSet [24]	Supporting EnumerableSet in Solidity.
Math [25]	Math-related utilities for Solidity.
MerkleProof [26]	Toolkit of Merkle Tree proofs.
AddressUpgradeable [32]	Address for upgradeable contracts.

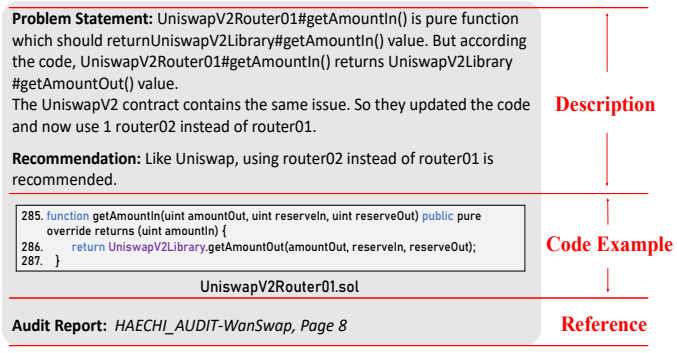

Figure 2: Card example for categorizing library misuse

As shown in Figure 2, a card contains three data fields: a misuse description, a code example for comprehension, and the audit report reference. In this example, the two researchers find that this card is related to the incorrect use of a library function, as the card description indicates that the contract invoked the *getAmountOut* function instead of the *getAmountIn* function, which resulted in an unexpected return value. Based on the referenced audit report, they further confirm the specific reason is that the contract developer ignored the handing fee calculation logic in the *getAmountOut* function. *UniswapV2Library* is a top 50 popular library, the researchers also find other similar cards, and grouped them together into a single category. Subsequently, according to the misuse reason, both researchers concurred to designate this category as "Underestimated Library Capability".

Subsequently, the author organizes another meeting with the same two volunteers to discuss any discrepancies. After reaching a consensus on category names, the author compares the differences between the classification results from two volunteers, and evaluates the similarity with Cohen's Kappa index [49]. The overall Kappa value is 0.89, indicating strong agreement. Then, the author and two volunteers vote for each different case and determine a consistent result. Finally, we identify eight library misuse patterns from 92 cases. These patterns cover the complete lifecycle of the library from development to utilization.

4 LIBRARY MISUSE PATTERNS

We find 8 patterns of library misuse through open card sorting. In this section, we give the definition of the patterns, and illustrate the cause and consequence of each pattern with several examples of different types of libraries.

4.1 P1: Invalid Wrapper Check in Library

Pattern introduction: This pattern occurs if an invalid check is implemented in the wrapper functions during library development. Many popular libraries are specifically designed for security purposes, with wrapper functions that implement additional checkers over original operations to prevent potential attacks. For instance, the SafeMath library [29] performs extra result checks on arithmetic operations to protect against integer overflow, the SafeCast library [27] performs extra input length checks on type convert operations to protect against illegal conversion.

In Solidity, the additional checking logic of wrapper functions is typically implemented using the "require" statement which enables the checking of conditions before executing a protected code fragment. If the specified condition is not met, it reverts the entire transaction. Library developers need to design appropriate conditions for critical operations. If these conditions are incorrect to cover all situations, they may fail to provide the necessary protections.

Example: SafeERC20 is a wrapper library over safe ERC-20 token operations [43], as there are several potential risks when developers handle token (such as missing return value check and supporting token with no return value). In this example, we focus on the *approve* operation, which can be used to approve a certain number of tokens for the spender from caller.

As shown in Listing 1, the *approve* function have a vulnerability which may lead to an unexpected allowance amount. A contract cannot modify the allowance by recalling the *approve* function with a non-zero amount, as if the spender spends the allowance provided by the first approval, the second approval is not a modification of the first approval allowance, but rather an additional allowance is issued. The library function *safeApprove* is the wrapper over the original *approve* function, which limit the allowance can only be set from zero, or be changed to zero.

```
1 library SafeERC20 {
2   function safeApprove(IERC20 token, address spender,
        uint256 value) internal {
3     require((value == 0) }} (token.allowance(msg.sender,
          spender) == 0)); // ERROR
4     require(token.approve(spender, value));
5 }}
```

Listing 1: Unfulfilled library implementation

To address this issue, SafeERC20 uses *safeApprove* to replace *approve*, which is a fixed approve function with more limitation in usage conditions. *safeApprove* only allows a contract to set the allowance from zero or change the allowance to zero. Changing allowances is suggested to use *safeIncreaseAllowance* and *safeDecreaseAllowance* instead of resetting the allowance value directly.

The library limits the usage of approve at line 3. Before invoking the approve function, *safeApprove* requires the target allowance value to be zero, or *msg.sender*'s allowance to be zero. However, the library developer might not be familiar with the conceptions of *msg.sender* and *address(this)*, which makes the checking is invalid. *msg.sender* represents the address of the transaction sender, and *address(this)* represents the address of the smart contract imported into this library. Hence, *safeApprove* should check *address(this)* but not *msg.sender*. Missing this limitation logic makes *safeApprove* can directly reset the allowance, which is not allowed according to the design goal. This example [13] is found in openzeppelin contracts before commit fc17a1d [16].

4.2 P2: Unhandled Exceptions in Library

Pattern introduction: This misuse pattern occurs when the library lacks exception handling. During library development, the library developer may not consider the robustness and security requirement in Solidity, and only implement the core business feature.

Handling all exceptions can be difficult in certain contexts. Libraries may involve interactions with other contracts, and the response from external contracts might be uncontrollable and varied. There are also some exceptions that are covert, such as appearing only in specific compiled versions. Furthermore, contract maintainers may update the compiler version of the library, which may also introduce new pending exceptions (e.g., Openzeppelin keeps updating the compiler version in its repository [19]. By January 2023, all libraries have been updated to upper 0.8 version).

Example: IERC165 is a Solidity interface specified in the ERC-165 standard [12]. This interface provides a standard mechanism for querying whether a contract supports a particular interface (e.g., ERC20 token interface). To comply with IERC165, all supported standard interfaces should be declared in the *supportsInterface* function. Notably, there are two steps to query the interfaces supported in a smart contract. First, caller contracts need to confirm whether the target contract implements IERC165. Then, if IERC165 is implemented, the caller contracts can invoke the *supportsInterface*

function of the target contract to get the result of the supported interfaces. The ERC165Checker library is specifically designed to implement the first step for caller contracts. Library function *supportsERC165InterfaceUnchecked* achieves this by constructing a *STATICCALL* to the target contract and using the returned value as the basis for its determination.

```
1 library ERC165Checker {
2   function supportsERC165InterfaceUnchecked(address
        account, bytes4 interfaceId) returns (bool) {
3     bytes memory encodedParams = abi.encodeWithSelector(
          IERC165.supportsInterface.selector,interfaceId);
4     (bool success, bytes memory result) = account.
          staticcall{gas: 30000}(encodedParams);
5     if (result.length < 32) return false;
6     return success && abi.decode(result, (bool));
7 }}}
```

Listing 2: Vulnerable library implementation

In this example, ERC165Checker makes a static call at line 4, and the row data is decoded to bool type at line 6. In Solidity 0.8.0 and later versions, the *abi.decode* function will revert if the raw data overflows the target type. Hence, if the decoded data occupies more than one byte, the bool variable declared at line 6 will overflow and cause a revert, which introduces an unhandled exception. A malicious contract can construct a *supportsInterface* function with a return value exceeding one byte, resulting in rollbacks of callers, which may lead to financial loss in auction and gambling scenarios.

4.3 P3: Inappropriate Library Extension

Pattern introduction: This misuse pattern occurs when multiple independent features are implemented in one method. Typically, libraries are designed to offer a range of homogeneous operations (e.g., library SafeMath provides 13 functions to support different safe arithmetic operations). However, in this pattern, the library developer designs too many functionalities in a single function, and some of the functionality may not be used when the smart contract invokes this library function. The unused code introduces risks to the caller contract. Although contracts are not expected to execute these codes, they may still be executed under certain conditions (e.g., specific inputs), which may lead to unexpected consequences.

```
1 library ECDSA {
2   function recover(bytes32 hash, bytes memory signature)
        internal pure returns (address) {
3     bytes32 r; bytes32 s; uint8 v;
4     // Code omitted: Calculating r, s, v from signature
5     if (v < 27) { v += 27;} // ERROR
6     // Code omitted: Invoking ecrecover(hash, v, r, s)
7 }}
```

Listing 3: Inappropriate library extension

Example: The ECDSA library is used to recover the address of the signature. In this example, we focus on *recover* function, which can recover signatures containing *r*, *s*, *v*. The *v* is a bool value, which is confined to either 27/28 version format or 0/1 version format[1]. These two formats are both acceptable, while 27/28 is more prevalent, since this format is explicitly mentioned in the Ethereum yellow paper [70]. As shown in Listing 3, *ecrecover* originally supported the 27/28 format *v* as the input at line 6. However, in order to extend the support for another 0/1 format *v*, this function adds the logic in the original function at line 5, instead of creating a new function that only supports format 0/1 *v*. For contracts that do not use 0/1 format *v*, the expansion at line 5 is not only redundant, but also risky. If the contract developer uses only (*r*, *s*, 0/1) as the signature format, the unused format conversion code at line 5 can still be exploited by adversaries, as they can forge a (*r*, *s*, 0/1) signature based on a previous (*r*, *s*, 27/28). The example code can be found in openzeppelin-contract before committing 547a5f2 [15].

[1]In ECDSA, the v value is a 1-byte value that serves to identify the key used for signing. Specifically, v is set to 27 or 28, depending on whether the key used for signing corresponds to the lower or higher half of the secp256k1 curve. Additionally, the v can be represented in an alternative format, 0/1, which bears an equivalent meaning to the corresponding values of 27/28.

4.4 P4: Inappropriate Using For

Pattern introduction: In Solidity, the suggested way to utilize a library in a contract is by declaring a *using L for T* statement, where all non-private functions of library *L* will be attached to the data type *T* as member functions, and the member functions receive the object of type *T* as their first parameter. Notably, declaring a *using for* statement for a data type that is unsupported in the library will not trigger a warning from the compiler, since the type is only checked when the function is called. This *using for* statement affects code readability, as the variables of the unsupported type can only invoke internal library functions after forced type conversion. Furthermore, forced type conversion during downcasting is unsafe, which may introduce risks to the smart contract.

```
1 contract ExampleContract {
2   using SafeMath for uint256;
3   using SafeMath for int256;  // ERROR
4   function calculate(int a, uint b) public returns (
        uint256){
5     return uint256(a).div(b);  // Forced type convert
6 }}
```

Listing 4: Inappropriate using for

Example: The SafeMath library is utilized in Solidity to avoid integer overflow in uint256. Uint256 is an unsigned integer data type with a width of 256 bits. In contrast, int256 is a signed integer data type that supports negative values.

As shown in Listing 4, the *using for* statement at line 3 is useless for int256, since the SafeMath library can only prevent integer overflow for uint256. Hence, variable *a* can only invoke *div* after being converted to uint256 at line 5, which can also be executed without the *using for* statement at line 3. In this example, the useless *using for* statement is confusing, which decreases the readability of the smart contract. We find several similar misused *using for* statements in multiple real-world contracts, including those that occur in other libraries. In addition, it is not recommended to invoke libraries by forcing type conversions, as it is unsafe in Solidity.

4.5 P5: Incomplete Function Replacement

Pattern introduction: Certain libraries are designed as wrappers over original Solidity operations to enhance contract security. In such cases, failure to replace any original operations may result in vulnerabilities being introduced into the contract. In the last pattern, we demonstrate the use of the *using for* statement, which attaches all library functions to the target type. Although the *using for* statement could simplify the library use, developers may still fail to fully replace original operations with library methods, as library

functions may have similar names to the original functions. For example, in the SafeERC20 library, the *token.safeApprove* function is similar to the original *token.approve* function.

```
1  contract BasicToken is ERC20Basic {
2    using SafeMath for uint256;  // ... }
3  contract StandardToken is ERC20, BasicToken { //... }
4  contract PausableToken is StandardToken, Pausable {
5    function batchTransfer(address[] _receivers, uint256
         _value) public whenNotPaused returns (bool) {
6      uint cnt = _receivers.length;
7      uint256 amount = uint256(cnt) * _value;  // ERROR
8      for (uint i = 0; i < cnt; i++) {
9        balances[_receivers[i]] = balances[_receivers[i]].
             add(_value);
10 }}}
```

Listing 5: Incomplete function replacement

Example: BeautyChain [1] is a DApp known as the victim of the integer overflow attack in April 2018. As shown in Listing 5, an unsafe operator is used in the PausableToken contract at line 7, which can lead to the overflow of the variable *amount*. Hackers exploited this vulnerability and transferred enormous amounts of BEC tokens, resulting in a loss of over 1,000,000,000 dollars.

Some reports [46, 62] attribute that developers should increase the awareness of avoiding integer overflow and use the SafeMath library. However, contract PausableToken inherits contract BasicToken, which has introduced the SafeMath library at line 2. This means the wrapper functions in the SafeMath library are also inherited in the contract PausableToken. In addition, we noticed that in this example, the wrapper function *add* has been utilized at line 9 to replace just one original operation. Hence, avoiding incomplete function replacement is more critical in this example.

4.6 P6: Overestimated Library Capability

Pattern introduction: This pattern occurs when contract developers successfully invoke a library function but overestimate its capability. Typically, features and usage scope of a library are given through documentation (e.g. openzeppelin docs [20]), but sometimes this does not ensure that contract developers can fully understand the capabilities of the library. Exaggerating the capabilities of a library can lead to vulnerabilities, as some operations expected by users are not implemented by invoking library functions. Noticeably, P6 only occurs when the contract successfully invokes and executes the target library function, while P4 and P5 occur when the contract does not successfully invoke the target function due to useless *using for* statements or incorrect invoking function names.
Example: Fei Protocol [9] is a DApp supporting an algorithmic stable coin. Algorithmic stable coins could be attacked by the flashloan [69], which is usually implemented through smart contracts. Therefore, Fei Protocol is designed to refuse invokes from smart contracts to ensure its security.

```
1  contract BondingCurve{
2    function allocate() external {
3      require((!Address.isContract(msg.sender)) || msg.
           sender == core().genesisGroup(), "BondingCurve:
           Caller is a contract"); // ERROR
4      // ...
5  }}
```

Listing 6: Overestimated library capability

As shown in Listing 6, *BondingCurve* is a core contract in the Fei Protocol, and library function *isContract* is used to refuse contract invokes at line 3. However, the capacity of the function *isContract* is exaggerated in this example, as it only checks whether the length of the running code is empty, while the length of the construct function is not considered. Therefore, the *isContract* function at line 3 can not be used to refuse a contract that only has a construct function. The example code can be found in commit d8aebc2 [10]. This issue [11] has been reported as a common flash loan attack vulnerability, and it is caused by exaggerating the library capability.

4.7 P7: Underestimated Library Capability

Pattern introduction: This pattern represents the opposite situation of overestimated library capability. Contract developers may successfully invoke a library function but ignore the partial logic of the library function, and introduce unnecessary operations in the smart contract. These additional operations may throw exceptions during invoking, or return an unexpected value to the contract.

```
1  import './libraries/UniswapV2Library.sol';
2  contract UniswapV2Router01 is IUniswapV2Router01 {
3    function getAmountOut(uint amountIn, uint reserveIn,
         uint reserveOut) public returns (uint amountOut) {
4      return UniswapV2Library.getAmountOut(amountIn,
           reserveIn, reserveOut);
5    }
6    function getAmountIn(uint amountOut, uint reserveIn,
         uint reserveOut) public returns (uint amountIn) {
7      return UniswapV2Library.getAmountOut(amountOut,
           reserveIn, reserveOut); // ERROR
8  }}
```

Listing 7: Underestimated library capability

Example: UniswapV2Router01 [3] is a router contract, which provides exchange rate calculation service of the Uniswap exchange. For a token pair TokenA / TokenB, when exchanging token A for token B (expressed as $TokenA \rightarrow TokenB$) by specifying the amount of token A held, function *getAmountOut* calculates the corresponding quantity of token B that can be obtained. Conversely, by specifying the amount of token B expected, function *getAmountIn* calculates the amount of token A required. These two features have been implemented in the UniswapV2Library, and the router contract only needs to invoke the corresponding library functions.

As shown in Listing 7, the function *getAmountOut* is implemented by invoking the identically named library function correctly at line 4, while function *getAmountIn* is also implemented by invoking library function *getAmountOut* in a reverted exchange scenario ($TokenB \rightarrow TokenA$), which is feasible for token exchange without handling fees. However, the developer ignores the logic that the library function also calculates additional handling fees. The library function returns the value after deducting the transaction fee, causing the router function *getAmountIn* to return a value smaller than expected. However, since the handling fee is only 0.3%, this calculation error is difficult to recognize. Currently, UniswapV2Router01 [44] has been deprecated after providing service in over 233,000 transactions.

4.8 P8: Unnecessary Library Using

Pattern introduction: This pattern occurs when unnecessary libraries are used in smart contracts. With the update of Solidity,

the compiler has been equipped with more security checks. Consequently, it has become unnecessary to use corresponding security libraries in smart contracts. Although using such libraries may not cause errors, it can result in unnecessary gas consumption.

Example: From Solidity v0.8, the compiler automatically conducts overflow checks for arithmetic operations, obviating the need for the SafeMath. However, several popular libraries (e.g., Counters, SafeERC20) also use the SafeMath, and refactoring all such libraries can be challenging. In this situation, contract developers can import the SafeMath from openzeppelin contracts [29], which has removed the extra high-gas-consumed checks in the SafeMath.

5 LIBRARY MISUSE IN REAL-WORLD

We have identified eight patterns of library misuse based on audited contracts. However, audited reports are usually associated with high-value contracts compared to normal real-world contracts, and patterns concluded from those special audited contracts may not apply to real-world contracts. Hence, we need an additional investigation to understand library misuse in the real world.

We construct a dataset from 156,761 real-world contracts that used libraries. To ensure that the dataset accurately represents the real-world scenario, we first calculate the usage frequency of the top 100 libraries in the real-world contracts, as these libraries can cover over 351,926 (90%) library usage cases (c.f. Section 3.2). Then, we randomly select contracts and add them to our dataset to ensure that the usage proportion of each top 100 libraries in the selected dataset matches the real-world scenario. Finally, we obtain a dataset with 1018 contracts, which can reflect the usage distribution of the top 100 popular libraries in real-world contracts.

We hire two volunteers with over two years of experience in smart contract development to analyze our dataset. The volunteers are asked to label each case with the corresponding misuse pattern. If a misuse case belongs to one of the eight patterns, the volunteers use the corresponding pattern name as the label. If it cannot be categorized into the existing patterns, the volunteers label it as "unknown". Then, the author compares the detection results and organizes a meeting with two volunteers. During the meeting, the author identifies any missing or inconsistent cases in the results. The author and the two volunteers then vote to reach a consensus.

5.1 Universality of Misuse Patterns

To analyze whether our patterns are also prevalent in real-world contracts, we compare the number of misuse cases found in audit reports and real-world contracts. As shown in Table 2, all eight patterns can be identified within the 456 real-world contracts, accounting for a total of 905 cases of misuse (***Finding 1***). Noticeably, we find 71 contracts containing multiple pattern misuses, which results in the total contract number being smaller than the simple sum of the contract number of individual patterns. Compared to the 92 cases identified in 86 audited contracts, the average number of misuse cases found in real-world contracts is higher than that in audited contracts. In both audited and real-world contracts, Incomplete Function Replacement (P5) and Exaggerated Library Capability (P6) are the most frequently occurring patterns. In contrast, patterns pertaining to library implementation (P1, P2, P3) are relatively rare. Generally, the patterns found in audited contracts widely exist in real-world smart contracts.

Table 2: The number (#) of different library misuses patterns in audited contracts and real-world contracts.

Pattern	Audited contract		real-world contract	
	# Case	# Contract	# Case	# Contract
P1	5	5	1	1
P2	6	6	4	4
P3	3	3	2	2
P4	4	4	3	3
P5	36	30	543	204
P6	18	18	85	51
P7	8	8	1	1
P8	12	12	266	266
Total	92	86	905	456

Finding 1: *Our eight misuse patterns are practical, and real-world misuse cases can be categorized into our patterns.*

5.2 Frequency of Subtle Patterns

Our study reveals that certain misuse patterns (P5, P6) frequently occur in both audited and real-world contracts, suggesting that they might be challenging for developers to identify. We find these patterns may be triggered multiple times within a single contract. Considering the complexity of contracts, which typically consist of over hundreds of lines, avoiding this type of misuse in multiple code locations may be challenging. In this paper, we define these patterns as the *subtle patterns*. As an illustration, Incomplete Function Replacement (P5) is a subtle pattern, as it requires developers to recognize the difference between original and library functions and use library functions as wrappers to replace original functions.

Furthermore, the misuse frequency of these patterns (P5, P6) is considerably higher in real-world contracts than in audited reports. In real-world contracts, 543 (60%) misuse cases are attributed to P5, while the corresponding number is only 36 (39%) in audit reports.

The reason for this inconsistency is that more real-world contracts contain multiple P5 misuse cases, resulting in a higher average number of misuse cases found within each contract. Among the real-world contracts, 93 (45%) of them contain multiple P5 misuse cases, leading to 432 (80%) real-world cases. For example, POSC [2] is a selected real-world contract with over 100 transactions, and presents various P5 issues due to its use of multiple arithmetic operators without the corresponding SafeMath functions in 10 different positions, which lead to five functions and one modifier at risk of overflow. In contrast, only 6 (20%) audited contracts contain multiple P5 misuse cases. This evidence suggests that these real-world misuses may not be made accidentally, but rather originate from a lack of security awareness among the developers (***Finding 2***). Therefore, real-world developers may need to more systematically understand patterns of library misuse.

Finding 2: *Some real-world developers are unaware of subtle mistakes, which may lead to multiple similar misuses in one contract.*

5.3 Underestimated Capabilities in Libraries

We identify a total of nine cases of Underestimated Library Capability (P7) misuse in audited contracts and real-world contracts. The underestimated library capabilities in these cases are all related to arithmetic calculations (***Finding 3***). The developer may have neglected arithmetic calculations in the library function, resulting in an unexpected output. To assist developers in avoiding such library misuse, it is critical to analyze this phenomenon. We compare the differences between arithmetic operations with other operations in the library and analyze why arithmetic operations are more prone to being underestimated.

Finding 3: *Capabilities related to arithmetic calculations are more prone to be neglected, contract developers should be careful about these operations to avoid library misuse.*

The change in contract state after executing an arithmetic operation might be harder to be noticed by the caller compared to that caused by other operations. When developers may not review the entire library source code carefully, operations in the library can be regarded by them. As shown in Figure 3, we compare the contract state after executing different operations. In the first situation, the developer neglects partial logic (e.g., access check, token transfer), which leads to throwing errors or emitting messages. These consequences can be directly observed outside of the library function. In the second situation, the neglected arithmetic operation only manipulates the return value of the library function. If the return value is a boolean (0 or 1), the developer can easily identify the exception. However, if the return value is an unsigned integer (0 to $2^{256}-1$), it may be difficult for the developer to be aware of this unexpected value. Therefore, contract developers need to carefully review the source code or API documentation of the library, and exercise caution with arithmetic operations to prevent misuse.

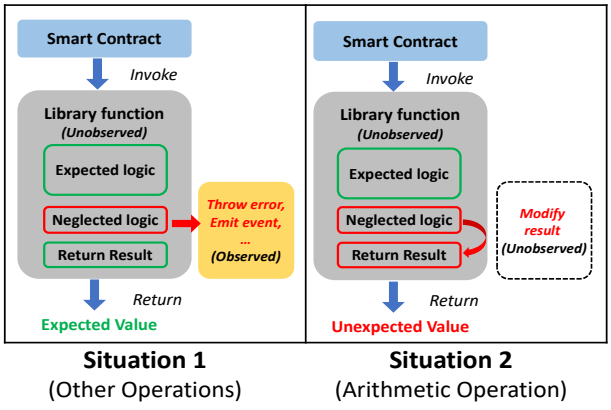

Figure 3: Comparison of different neglected logic in P7

5.4 Neglected Impact

We find 266 contracts related to Unnecessary Library Using (P8), which will not lead to vulnerabilities but only extra gas consumption. We investigate this neglected impact, which may deserve more attention from developers. As an illustration, we examine the extra gas consumption of the SafeMath library. We examine the extra gas consumption in the Remix and find each library function will waste 210 gas units on average. In our selected contracts, ShibAsia [37] contains a token transfer function, invoking SafeMath functions over 1120 times, resulting in over 1,123,500 extra gas waste. Although some methods (e.g., listing unnecessary libraries in Solidity Docs [4]) have been proposed, the issue still needs to be further addressed, as similar cases can be found in 266 (26%) selected contracts. In addition, gas waste will keep increasing with the number of transactions. Hence, P8 is a long-term and wide-ranging issue that should not be underestimated (***Finding 4***).

Finding 4: *Misuse pattern P8 exists in many real-world contracts. Although this pattern does not result in direct security issues, it still can lead to other long-term negative impacts.*

5.5 Propagation of Using Vulnerable Libraries

We observe two special P3 cases in real-world contracts which were deployed after 2021 but still used some earlier vulnerable library versions, although the patched versions had already been released in 2020 [41]. Similar issues may also exist in P1 and P2, since these patterns all lead to vulnerable libraries. Therefore, we investigate the vulnerable libraries related to P1, P2, and P3.

In order to investigate the longer-term library propagation, we supplement with early Ethereum contracts from 2015 to 2019 from XBlock-ETH [72], since the 223,336 real-world contracts obtained based on Smart Contract Sanctuary are mainly distributed from 2019 to 2022. Finally, we collect 468,744 Ethereum contracts since 2015. Then, we extract the library code used by each contract according to the abstract syntax tree (AST). Following this, we compare the code differences between the known vulnerable library and its corresponding fixed library and extract code snippets that only existed in the vulnerable library (e.g., invalid checks) as text features to identify vulnerable libraries. Regular expression matching is a method to match the partial text in the document, we construct corresponding regular expressions to match each text feature of the vulnerable library, and perform the matching for each library extracted from the real-world contracts. The resulting matches are manually scrutinized to ensure contextual relevance within the corresponding library.

As shown in Figure 4, we present three timelines depicting the propagation of each pattern-related library over the past seven years. P2 occurs only when the library fails to handle new exceptions, thus concentrating on major updates of Solidity in recent years. Vulnerable libraries continue to spread, although fixed versions have been released. Our dataset encompasses Ethereum contracts before December 2022, and we do not find a cessation of propagation. For example, the fixed time of P1-related libraries is February 2019, but we can still find various related vulnerable libraries deployed after this time, which means the vulnerable library has been spreading for nearly three years (***Finding 5***).

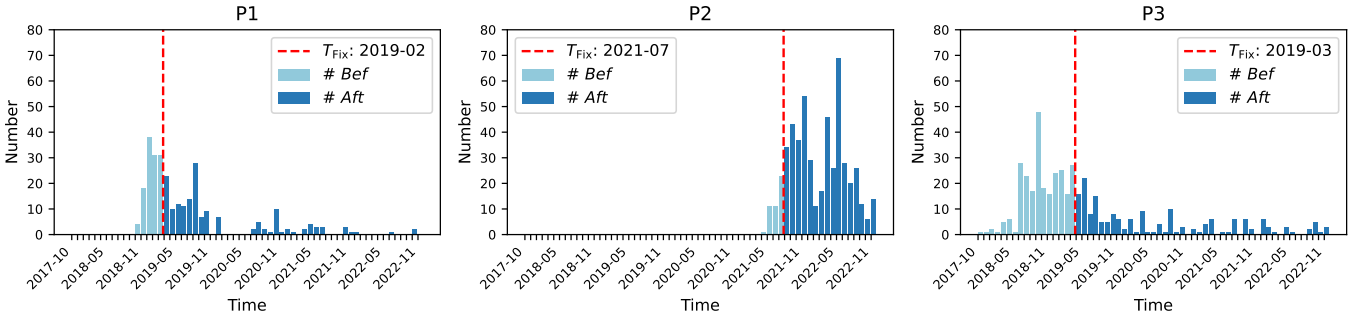

Figure 4: Timeline of vulnerable libraries propagation. # Bef and # Aft denote the number of misuse cases per month before and after the fixed version is released. T_{Fix} denote the time of the fixed library version release.

> ***Finding 5:*** *Vulnerable libraries are still spreading after fixed library versions are released, which means P1, P2, and P3 are long-term risks to contract security.*

A library is called deprecated when the library maintainer stops maintaining it. For example, Openzeppelin replaced the ECRecovery [39] library with the ECDSA library, as they implemented similar functions. Both ECRecovery and ECDSA contain similar P3 issues, but only ECDSA is fixed by the developers, as ECRecovery has been deprecated. Releasing a fixed version can not entirely prevent the propagation of vulnerable libraries, but taking no action will only exacerbate the propagation (***Finding 6***). In pattern P3, 240 cases are found from deprecated libraries, while 197 cases are found from maintained libraries. We only find a few deprecated vulnerable libraries related to P3, but libraries related to P1 and P2 may also face this risk if they are deprecated in the future.

> ***Finding 6:*** *Misuse patterns such as P3 in deprecated libraries may lead to wider propagation than that in maintained libraries.*

5.6 Suggestions for Developing Detection Tools

Based on the characteristics of each misuse pattern, there are some suggestions for developing library misuse detection tools. For vulnerable libraries (P1, P2, P3), the detection tool not only needs to detect the potential vulnerabilities (such as integer overflow, unhandled return value, etc.) but also needs to focus on the coverage of library functions, as these functions are only executed after being invoked by contracts. For Inappropriate Using For (P4), the detection tool needs to compare the return value types of the library functions with those in the using for statements. For Incomplete Function Replacement (P5), the tool may need to compare the differences between library wrapper functions and the original functions. For patterns related to library capabilities (P6, P7), some formal methods may be used to define library capabilities.

6 THREATS TO VALIDITY

6.1 Internal Validity

We established a strategy for selecting audit services by December 2022, excluding companies with a small number (< 10) of published reports and those that do not open source their reports. The reports collection time and strategy made us miss certain audit reports. However, the 593 collected audit reports already include many valid cases of library misuse, which can reflect various different situations of library misuse in audit reports. We hired two volunteers to manually label misuse cases from 1,018 real-world contracts. The labeling quality may be affected by the volunteers' personal experience and the complexity of real-world contracts. However, the results from the two volunteers were cross-validated, and inconsistent results were verified to circumvent false positives cases, which can ensure the quality of the dataset.

6.2 External Validity

Our patterns are extracted from audit reports, so our patterns are limited to the knowledge contained within those audit reports. However, audit reports are authored by security experts from reputable and experienced audit companies we selected. We utilized the Smart Contract Sanctuary dataset to obtain real-world Ethereum contract addresses. However, the contracts of this dataset are mainly distributed after 2019, so our analysis is more aligned with the library misuse in the recent three years.

7 RELATED WORK

7.1 Empirical Studies

With the development of Ethereum, the security of smart contracts is receiving increasing attention, and some empirical studies have been proposed to investigate blockchain and smart contract security. Zheng et al. [73] presented a survey on blockchain technology, and pointed out smart contracts as the future direction. They highlighted the potential financial risks associated with smart contract bugs and categorized smart contract researches into two types, i.e., contract development and contract evaluation. Chen et al. [47] defined 20 types of smart contract defects during contract development by analyzing smart-contract-related posts from Ethereum StackExchange and real-world smart contracts. This work can help developers better understand the symptoms of smart contract defects. Durieux et al. [50] presented an empirical study on smart contract evaluation methods. They investigated 9 automated detection tools on two datasets, revealing the challenges of smart contract vulnerability detection. Su et al. [66] investigated large-scale incidents and identified 476,342 hacking transactions on 85 targets DApps. Their work resulted in the creation of the largest

Ethereum on-chain DApp attack dataset. The above studies provide valuable insights into smart contract security. However, existing studies have not systematically investigated library misuse.

7.2 Automatic Tools

Several automatic tools have been developed to detect vulnerabilities in smart contracts. However, there is a lack of tools designed for detecting library misuse in smart contracts.

7.2.1 Static Analysis Tools: Static analysis tools analyze the logic or structure of contracts without executing code. Oyente [58] was the first tool to detect potential security issues in Ethereum smart contracts. It uses symbolic execution to generate a control flow graph and design pre-defined patterns to detect the issues. Slither [52] works by converting smart contracts into an intermediate representation. By utilizing extensive semantic information, Slither can effectively detect over 84 contract issues. There are many other vulnerability detection tools based on static analysis, e.g., Securify [68], MAIAN [61], Osiris [67], and SmartTest [64].

7.2.2 Fuzzing Tools: Fuzzing tools execute a smart contract with various inputs and monitor its behavior to uncover potential vulnerabilities. ContractFuzzer [57] utilized smart contract specifications, defining test oracles, instrumenting the EVM, and analyzing runtime behaviors to generate fuzz input, and successfully identified over 459 vulnerabilities among 6,991 tested smart contracts. The sFuzz [60] was inspired by C language fuzzers, and implied an efficient lightweight multi-objective adaptive strategy, achieving high code coverage. Moreover, some tools employ multiple techniques to detect vulnerabilities, such as The Imitation Learning Fuzzer [56], which learns from symbolic execution experts and swiftly applies that knowledge to its fuzzing approach.

7.2.3 Frameworks: Some frameworks have been proposed to improve the accuracy and efficiency of smart contract vulnerability detection. SODA [48] is a generic online detection framework for smart contracts on blockchains, facilitating the development of detection apps with ease, efficiency, and compatibility. TXSPECTOR [71] is a framework for investigating Ethereum transactions to detect attacks, allowing users to specify customized rules for attack detection. It has been evaluated for detecting three types of attacks and can be used for forensic analysis. Gigahorse [55] is a toolchain that decompiles Ethereum Virtual Machine bytecode into high-level code for smart contract analysis, offering high precision, completeness, and scalability.

8 CONCLUSION AND FUTURE WORK

In this paper, we focus on the issue of library misuse in smart contract development and utilization, which could lead to contract defects and financial losses. We conduct an empirical study by collecting over 500 audit reports and analyzing 223,336 smart contracts to identify general patterns of library misuse. Library misuse is a widespread issue, with 905 cases identified in real-world contracts. We conclude eight patterns of library misuse, which can help smart contract developers on Ethereum to prevent library misuse and potential financial losses. We publish the first dataset of library misuse, including 997 (92 + 905) library misuse cases from audit reports and real-world contracts.

In the future, efforts should be made to develop effective tools and methods to detect and prevent library misuse. This could involve creating automated analysis tools to identify general patterns of library misuse and integrating these tools into existing smart contract development workflows, which can help developers to avoid library misuse. In addition, more measures can be taken to improve the transparency and accessibility of library-related information for developers, retire vulnerable libraries more quickly and effectively, and establish a system for tracking library usage and corresponding vulnerabilities.

ACKNOWLEDGMENTS

This work is partially supported by fundings from the National Key R&D Program of China (2022YFB2702203), the National Natural Science Foundation of China (62032025, 62002393), Guangdong Basic and Applied Basic Research Foundation (2023A1515011336).

REFERENCES

[1] Apr., 2018. *Beauty Chain | Address 0xc5d105e63711398af9bbff092d4b6769c82f793d.* https://etherscan.io/address/0xc5d105e63711398af9bbff092d4b6769c82f793d
[2] Apr., 2018. *POSC | Address 0x3d807baa0342b748ec59aa0b01e93f774672f7ac.* https://etherscan.io/address/0x3d807baa0342b748ec59aa0b01e93f774672f7ac
[3] Apr., 2020. *Library UniswapV2Router01.* https://github.com/Uniswap/v2-periphery/blob/0335e8f7e1bd1e8d8329fd300aea2ef2f36dd19f/contracts/UniswapV2Router01.sol original-date: 2019-12-09T20:17:43Z.
[4] Apr., 2021. *Solidity v0.8.0 Breaking Changes — Solidity 0.8.19 documentation.* https://docs.soliditylang.org/en/v0.8.19/080-breaking-changes.html
[5] Aug., 2022. *Ethereum Development Documentation.* https://ethereum.org/en/developers/docs/smart-contracts/testing/#code-audits
[6] Dec., 2022. *Blockchain Consilium - REPORT VERIFICATION V2.* https://www.blockchainconsilium.com/verifyreport.html
[7] Dec., 2022. *CertK - Resources.* https://www.certik.com/resources/blog
[8] Dec., 2022. *Smart Dec Audit.* https://blog.smartdec.net/smart-contracts-security-audits/home
[9] Feb., 2021. *Fei Protocol.* https://github.com/fei-protocol/fei-protocol-core/blob/d8aebc2b119739ad1525d5c8861f2480d1610ddb/contracts/bondingcurve/BondingCurve.sol
[10] Feb., 2021. *fei-protocol BondingCurve.sol commit d8aebc2.* https://github.com/fei-protocol/fei-protocol-core/blob/d8aebc2b119739ad1525d5c8861f2480d1610ddb/contracts/bondingcurve/BondingCurve.sol
[11] Feb., 2021. *Pre release fix flash attacks by Joeysantoro · Pull Request #81 · fei-protocol/fei-protocol-core.* https://github.com/fei-protocol/fei-protocol-core/pull/81
[12] Jan., 2018. *ERC-165: Standard Interface Detection.* https://eips.ethereum.org/EIPS/eip-165
[13] Jan., 2018. *OpenZeppelin IERC20.sol.* https://github.com/OpenZeppelin/openzeppelin-contracts/blob/b7d60f2f9a849c5c2d59e24062f9c09f3390487a/contracts/token/ERC20/SafeERC20.sol
[14] Jan., 2019. *Ethereum.org.* https://www.ethereum.org/
[15] Jan., 2019. *OpenZeppelin ECDSA.sol commit 547a5f2.* https://github.com/OpenZeppelin/openzeppelin-contracts/blob/547a5f242a80c7df68015768c8770cc82a5e6058/contracts/cryptography/ECDSA.sol
[16] Jan., 2019. *OpenZeppelin SafeERC20.sol commit fc17a1d.* https://github.com/OpenZeppelin/openzeppelin-contracts/blob/fc17a1d9f58b7ca6e2de884769f8b6b362dc0e3c/contracts/token/ERC20/SafeERC20.sol
[17] Jan., 2023. *Ethereum Docs: Smart contract library.* https://ethereum.org/en/developers/docs/smart-contracts/libraries
[18] Jan., 2023. *Ethereum Docs: Smart contract library.* https://ethereum.org/en/developers/docs/smart-contracts/libraries/#related-tools
[19] Jan., 2023. *openzeppelin-contracts · Github.* https://github.com/OpenZeppelin/openzeppelin-contracts
[20] Jan., 2023. *OpenZeppelin Docs.* https://docs.openzeppelin.com/
[21] Jan., 2023. *Openzeppelin Docs - Address.* https://docs.openzeppelin.com/contracts/4.x/api/utils#Address
[22] Jan., 2023. *Openzeppelin Docs - Counters.* https://docs.openzeppelin.com/contracts/4.x/api/utils#Counters
[23] Jan., 2023. *Openzeppelin Docs - ECDSA.* https://docs.openzeppelin.com/contracts/4.x/api/utils#ECDSA

[24] Jan., 2023. *Openzeppelin Docs - EnumerableSet.* https://docs.openzeppelin.com/contracts/4.x/api/utils#EnumerableSet

[25] Jan., 2023. *Openzeppelin Docs - Math.* https://docs.openzeppelin.com/contracts/4.x/api/utils#Math

[26] Jan., 2023. *Openzeppelin Docs - MerkleProof.* https://docs.openzeppelin.com/contracts/4.x/api/utils#MerkleProof

[27] Jan., 2023. *Openzeppelin Docs - SafeCast.* https://docs.openzeppelin.com/contracts/4.x/api/utils#SafeCast

[28] Jan., 2023. *Openzeppelin Docs - SafeERC20.* https://docs.openzeppelin.com/contracts/4.x/api/token/erc20#SafeERC20

[29] Jan., 2023. *Openzeppelin Docs - SafeMath.* https://docs.openzeppelin.com/contracts/4.x/api/utils#SafeMath

[30] Jan., 2023. *Openzeppelin Docs - StorageSlot.* https://docs.openzeppelin.com/contracts/4.x/api/utils#StorageSlot

[31] Jan., 2023. *Openzeppelin Docs - Strings.* https://docs.openzeppelin.com/contracts/4.x/api/utils#Strings

[32] Jan., 2023. *Openzeppelin Github - AddressUpgradeable.* https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/utils/AddressUpgradeable.sol

[33] Jan., 2023. *OpenZeppelin/openzeppelin-contracts · Security Advisories Page.* https://github.com/OpenZeppelin/openzeppelin-contracts/security/advisories

[34] Jan., 2023. *PeckShield - Industry Leading Blockchain Security Company.* https://peckshield.com/#services

[35] Jan., 2023. *Smart Contract Sanctuary.* https://github.com/tintinweb/smart-contract-sanctuary

[36] Jan., 2023. *Smart Contracts Audit and Security.* https://etherscan.io/directory/Smart_Contracts/Smart_Contracts_Audit_And_Security

[37] Jul., 2021. *ShibAsia | Address 0x2d415bd832a37dd332a9b58c6a7ab209a2d1286c.* https://etherscan.io/address/0x2d415bd832a37dd332a9b58c6a7ab209a2d1286c

[38] Mar., 2018. *EtherScan.* https://etherscan.io/

[39] Mar., 2018. *openzeppelin-contracts ECRecovery.sol.* https://github.com/OpenZeppelin/openzeppelin-contracts/blob/e299a7950e31f35809339316dbbda894c6b52e01/contracts/ECRecovery.sol

[40] Mar., 2018. *Solidity Document.* http://solidity.readthedocs.io

[41] Mar., 2019. *OpenZeppelin/openzeppelin-contracts Pull Request #1622.* https://github.com/OpenZeppelin/openzeppelin-contracts/pull/1622

[42] Mar., 2023. *Dateset: library_misuse_data Github.* https://github.com/libraryMisuse/library_misuse_data

[43] Nov., 2015. *ERC-20: Token Standard.* https://eips.ethereum.org/EIPS/eip-20

[44] Sep., 2022. *Router01 | Address 0xf164fC0Ec4E93095b804a4795bBe1e041497b92a.* https://etherscan.io/address/0xf164fC0Ec4E93095b804a4795bBe1e041497b92a

[45] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272).* IEEE, 368–377.

[46] CertiK. Apr., 2018. *How Formal Verification Would Have Fortified Beauty Chain (BEC) Contract.* https://medium.com/certik/how-formal-verification-would-have-fortified-beauty-chain-bec-contract-f53e78159400

[47] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2020. Defining smart contract defects on ethereum. *IEEE Transactions on Software Engineering* 48, 1 (2020), 327–345.

[48] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, et al. 2020. SODA: A Generic Online Detection Framework for Smart Contracts.. In *NDSS*.

[49] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[50] 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.* 530–541.

[51] Yuanrui Fan, Xin Xia, David Lo, Ahmed E Hassan, Yuan Wang, and Shanping Li. 2021. A Differential Testing Approach for Evaluating Abstract Syntax Tree Mapping Algorithms. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* IEEE, 1174–1185.

[52] 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, 8–15.

[53] W.B. Frakes and Kyo Kang. 2005. Software reuse research: status and future. *IEEE Transactions on Software Engineering* 31, 7 (2005), 529–536. https://doi.org/10.1109/TSE.2005.85

[54] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. 2000. *The Java language specification.* Addison-Wesley Professional.

[55] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* IEEE, 1176–1186.

[56] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security.* 531–548.

[57] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* 259–269.

[58] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security.* 254–269.

[59] M Douglas McIlroy, J Buxton, Peter Naur, and Brian Randell. 1968. Mass-produced software components. In *Proceedings of the 1st international conference on software engineering, Garmisch Pattenkirchen, Germany.* 88–98.

[60] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering.* 778–788.

[61] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th annual computer security applications conference.* 653–663.

[62] p0n1. Apr., 2018. *A disastrous vulnerability found in smart contracts of BeautyChain (BEC).* https://medium.com/secbit-media/a-disastrous-vulnerability-found-in-smart-contracts-of-beautychain-bec-dbf24ddbc30e

[63] Martin Reddy. 2011. *API Design for C++.* Elsevier.

[64] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution.. In *USENIX Security Symposium.* 1361–1378.

[65] Donna Spencer. 2009. *Card sorting: Designing usable categories.* Rosenfeld Media.

[66] Liya Su, Xinyue Shen, Xiangyu Du, Xiaojing Liao, XiaoFeng Wang, Luyi Xing, and Baoxu Liu. 2021. Evil Under the Sun: Understanding and Discovering Attacks on Ethereum Decentralized Applications.. In *USENIX Security Symposium.* 1307–1324.

[67] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference.* 664–676.

[68] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* 67–82.

[69] Dabao Wang, Siwei Wu, Ziling Lin, Lei Wu, Xingliang Yuan, Yajin Zhou, Haoyu Wang, and Kui Ren. 2021. Towards a first step to understand flash loan and its applications in defi ecosystem. In *Proceedings of the Ninth International Workshop on Security in Blockchain and Cloud Computing.* 23–28.

[70] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* (2014).

[71] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. 2020. TXSPECTOR: Uncovering attacks in ethereum from transactions. In *USENIX Security Symposium.*

[72] Peilin Zheng, Zibin Zheng, Jiajing Wu, and Hong-ning Dai. 2020. XBlock-ETH: Extracting and Exploring Blockchain Data from Ethereum. *IEEE Open Journal of the Computer Society* 1 (2020), 95–106. https://doi.org/10.1109/OJCS.2020.2990458

[73] 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, 4 (2018), 352–375.