Using My Functions Should Follow My Checks: Understanding and Detecting Insecure OpenZeppelin Code in Smart Contracts

Han Liu, *East China Normal University, Shanghai Key Laboratory of Trustworthy Computing;* Daoyuan Wu, *The Hong Kong University of Science and Technology;* Yuqiang Sun, *Nanyang Technological University;* Haijun Wang, *Xi'an Jiaotong University;* Kaixuan Li, *East China Normal University, Shanghai Key Laboratory of Trustworthy Computing;* Yang Liu, *Nanyang Technological University;* Yixiang Chen, *East China Normal University, Shanghai Key Laboratory of Trustworthy Computing*

https://www.usenix.org/conference/usenixsecurity24/presentation/liu-han

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Using My Functions Should Follow My Checks: Understanding and Detecting Insecure OpenZeppelin Code in Smart Contracts

Han Liu[1*], Daoyuan Wu[2†], Yuqiang Sun[3], Haijun Wang[4], Kaixuan Li[1*], Yang Liu[3], and Yixiang Chen[1]

[1]*East China Normal University, Shanghai Key Laboratory of Trustworthy Computing*
[2]*The Hong Kong University of Science and Technology*
[3]*Nanyang Technological University*
[4]*Xi'an Jiaotong University*

Abstract

OpenZeppelin is a popular framework for building smart contracts. It provides common libraries (e.g., SafeMath), implementations of Ethereum standards (e.g., ERC20), and reusable components for access control and upgradability. However, unlike traditional software libraries, which are typically imported as static linking libraries or dynamic loading libraries, OpenZeppelin is utilized by Solidity contracts in the form of source code. As a result, developers often make custom modifications to their copies of OpenZeppelin code, which may lead to unintended security consequences.

In this paper, we conduct the first systematic study on the security of OpenZeppelin code used in real-world contracts. Specifically, we focus on the security checks in the official OpenZeppelin library and examine whether they are faithfully enforced in the relevant OpenZeppelin functions of real contracts. To this end, we propose a novel tool named ZepScope that comprises two components: MINER and CHECKER. First, MINER analyzes the official OpenZeppelin functions to extract the facts of explicit checks (i.e., the checks defined within the functions) and implicit checks (i.e., the conditions of calling the functions). Second, based on the facts extracted by MINER, CHECKER examines real contracts to identify their OpenZeppelin functions, match their checks with those in the facts, and validate the consequences for those inconsistent checks. By overcoming multiple challenges in developing ZepScope, we obtain not only the first taxonomy of OpenZeppelin checks but also the comprehensive results of checking the top 35,882 contracts from three mainstream blockchains.

1 Introduction

Smart contracts [9], software programs executed on blockchains, have gained prominence in recent years, finding applications in decentralized finance (DeFi) [54] and non-fungible tokens (NFTs) [25] among others. Given the immutable nature of blockchains, once deployed, smart contracts cannot be altered, persisting as long as the platform remains active. This immutability implies that vulnerabilities cannot be repaired without deploying a revised version as a new contract. Security pitfalls in smart contracts can result in substantial financial losses, as evidenced by the $7 million loss from the insufficient_check attack [2] in 2023. Thus, ensuring smart contract security prior to deployment is imperative.

To foster code standardization and mitigate security issues, OpenZeppelin [15] provides common libraries like SafeMath, implementations of Ethereum standards such as ERC20 [6], and reusable components for access control and upgradeability. A study [34] reveals that 36.3% of verified contracts employ OpenZeppelin code. Unlike traditional programming environments where libraries are imported via static or dynamic linking, in Solidity, OpenZeppelin code is typically cloned into contracts. Moreover, developers may alter or embed their logic into the code, potentially introducing unintended security consequences and substantial financial risks [18, 19].

In this paper, we aim to systematically investigate the security of OpenZeppelin code used in real-world contracts. Specifically, we focus on two types of security checks from the official OpenZeppelin library functions and examine whether they are faithfully enforced in the relevant OpenZeppelin function code of real contracts. One type of checks is *explicitly* defined within the functions and their callee functions, such as those in `requirement` and `modifier` statements. The other type of checks is *implicitly* enforced as the conditions of calling the functions, such as the default role checking when invoking `internal` OpenZeppelin functions. To scrutinize these checks, we propose a novel tool named ZepScope, comprising two components: MINER and CHECKER.

MINER analyzes the entire OpenZeppelin library to extract the facts of both explicit definition checks and implicit call checks. During this process, MINER addresses two challenges: (i) It performs an inter-procedural alias analysis to overcome the challenge that denifition check facts could manifest in various forms across the call chain; and (ii) It leverages code

[*] Work conducted by Han Liu and Kaixuan Li while they were visiting Ph.D. students at NTU.

[†] Corresponding author: Daoyuan Wu. Work conducted while at NTU.

context and data flow information to apply a relevance judgment for different call check facts that may intertwine within a single caller function. After running MINER, we obtain 1,435 OpenZeppelin check facts and organize them into a taxonomy with four major categories and three severity levels. More details about this taxonomy can be found in §4.6.

Based on the facts extracted by MINER, CHECKER detects insecure OpenZeppelin code in real-world contracts. Specifically, it first identifies the target OpenZeppelin-like functions within a contract through both the function signature and the contract structure information. CHECKER then extracts three types of checks, namely `require`, `if-revert`, and modifier checks, from the identified target functions. After that, CHECKER matches the extracted code checks with OpenZeppelin facts through a custom similarity measurement. Finally, CHECKER also validates the security consequences of the potentially insecure OpenZeppelin code by checking whether it could be actually exploited by an attacker.

To thoroughly evaluate ZepScope's effectiveness and usefulness, we conduct both a benchmark and a large-scale experiment. In the benchmark experiment with 51 real-world security bugs collected from multiple sources, we compare the effectiveness of fact-based detection employed by ZepScope and the pattern-based detection used by three state-of-the-art (SOTA) tools [27, 28, 31]. The results (detailed in §6.1) show that ZepScope significantly outperforms in detecting the vulnerability types that, although covered by SOTA tools, are related to OpenZeppelin checks. This superior performance is attributed to ZepScope's unique approach of understanding OpenZeppelin facts and leveraging them for detection.

In a large-scale experiment with 35,882 contracts from three major Ethereum-compatible blockchains, we evaluated ZepScope's accuracy and performance. ZepScope analyzed a total of 2,750,165 functions and identified 47,431 functions with potential insecure OpenZeppelin code, averaging about only one warning per contract. This is manageable for real-world manual inspection, especially as 39,225 warnings are low-level (e.g., missing zero address checks). Due to the lack of ground truth, we manually inspected a random sample of 100 warnings from each chain. Out of 300 warnings inspected, 31 were confirmed as false positives, yielding an accuracy of 89.67% across all sampled contracts. Moreover, ZepScope's average analysis time of 42.39 seconds per contract demonstrates its suitability for large-scale on-chain scanning.

The large-scale experiment also enabled us to uncover four notable security findings. First, by filtering and manually reviewing high-level warnings related to role checks, we identified 15 new vulnerabilities. Second, we discovered a prevalent absence of zero address checks in OpenZeppelin-related code, with 22,448 functions lacking this check. Although this issue does not trigger direct vulnerabilities, we present a threat model in §6.3, explaining how this common negligence can be exploited for effective phishing attacks. Third, we uncovered an interesting campaign involving 255 contracts on BSC alone, where OpenZeppelin checks are intentionally relaxed to serve their own logic. Lastly, we explore the differences in OpenZeppelin security checks across three chains; see §6.4.

Availability. We have made the code, dataset, and our evaluation results available at https://zepscope.github.io/.

2 Background

2.1 OpenZeppelin for Smart Contracts

OpenZeppelin [15] is one of the most popular packages for building secure smart contracts. It has a collection of high-quality reusable smart contracts that can be utilized to build decentralized applications (DApps) and protocols on Ethereum. For example, OpenZeppelin provides ERC20 [6] for fungible tokens, ERC721 [7] for non-fungible tokens, and ERC777 [8] for advanced tokens. These OpenZeppelin libraries continue to evolve and update to include new features and address security issues. The smart contracts in OpenZeppelin are written in Solidity, and the libraries also contain some interfaces, abstract, and virtual methods which can be modified and implemented by developers according to their needs. OpenZeppelin also provides a set of requirements for each API on how to avoid common security vulnerabilities, such as access control issues and arithmetic overflow and underflow, which could result in the loss of users' assets. However, not all developers adhere to these requirements, which has led to to security vulnerabilities in their smart contracts [18, 19].

2.2 A Motivating Example

Here we illustrate a real-world vulnerability from the Code4rena audit report [1], demonstrating how OpenZeppelin could be misused in a contract named NFTX. Figure 6 presents the original `flashLoan` function in `ERC20FlashMint` of OpenZeppelin, while Figure 1 depicts the vulnerable `flashLoan` function in NFTX, which implements a simplified version of OpenZeppelin's `flashLoan` function (specifically, without `flashFeeReceiver`). In the `flashLoan` function of `ERC20FlashMint`, the `amount` is kept strictly below `maxFlashLoan(token)` through an amount check in line 7, ensuring that the sum of `amount` and `fee` will not cause an overflow. However, the `flashLoan` function of NFTX lacks such a check and allows the message sender to control the amount. This creates an opportunity for an attacker to craft an amount to mint (via line 10), leading to a scenario where the sum of `amount` and `fee` overflows to a small value or even 0 (bypassing the check in line 13), indirectly enabling a large number of tokens to be minted while only a small portion of the amount is burned (i.e., line 14 and 15). Despite Solidity versions 0.8.0 and later having automatic checks for overflow and underflow, NFTX remains susceptible to this attack as it operates on Solidity version 0.6.8.

```
function flashLoan(
    IERC3156FlashBorrowerUpgradeable receiver,
    address token,
    uint256 amount,
    bytes memory data
) public virtual override returns (bool)
{
    // Vulnerable point: It misses the amount
        check in the original OpenZeppelin
        library; see Line 7 in Figure 6.
    uint256 fee = flashFee(token, amount);
    _mint(address(receiver), amount);
    require(receiver.onFlashLoan(msg.sender,
        token, amount, fee, data) ==
        RETURN_VALUE, "ERC20FlashMint: invalid
        return value");
    uint256 currentAllowance = allowance(
        address(receiver), address(this));
    require(currentAllowance >= amount + fee, "
        ERC20FlashMint: allowance does not
        allow refund");
    _approve(address(receiver), address(this),
        currentAllowance - amount - fee);
    _burn(address(receiver), amount + fee);
    return true;
}
```

Figure 1: The vulnerable `flashLoan` function in NFTX.

3 ZepScope Overview

In this section, we present an overview of ZepScope, the first tool designed for extracting OpenZeppelin facts and checking their violations in smart contracts. As depicted in Figure 2, ZepScope comprises two components: MINER and CHECKER. MINER is responsible for mining the facts of OpenZeppelin checks from the official OpenZeppelin library, and CHECKER is responsible for detecting insecure OpenZeppelin code in real-world contracts based on the facts mined by MINER. ZepScope has the following three major phases:

Firstly, we collect the OpenZeppelin library code from its official repository. More specifically, we used version 4.9.3, the latest version available when we initiated this study.

Secondly, in a one-time effort during the offline process, MINER analyzes the OpenZeppelin library to extract two types of facts regarding function checks. One is referred to as *Function Definition Facts*, while the other is called *Function Call Facts*. After MINER extracts these two kinds of checks, to enhance the accuracy and relevance of the extracted facts, we conduct a manual yet minor review on the facts and present a taxonomy of OpenZeppelin checks based on these facts.

Thirdly, during the online process, CHECKER detects insecure OpenZeppelin code in real-world contracts. Specifically, CHECKER first identifies the target functions within the contract. It then examines whether these target functions contain insecure OpenZeppelin code based on the facts mined by MINER. CHECKER also validates the security consequences of the potentially insecure OpenZeppelin code by checking whether it could be actually exploited by an attacker.

Next, we present the details of MINER and CHECKER in §4 and §5, respectively.

4 Mining OpenZeppelin Function Checks

4.1 The Challenges in Extracting Facts

As mentioned in §3, there are two types of facts regarding OpenZeppelin function checks, namely *Facts of Definition Checks* or $Fact_{Def}$ and *Facts of Call Checks* or $Fact_{Call}$. Without sacrificing clarity, we represent them using $Func_{caller}\{check_{caller}; Func_{target}();\}$, $Func_{target}\{check_{target}; Func_{callee}();\}$, and $Func_{callee}\{check_{callee}; Operation;\}$. For a given target OpenZeppelin function $Func_{target}$, its $Fact_{Def}$ is then defined as $check_{target} + check_{callee}$, while its $Fact_{Call}$ is represented by $check_{caller}$. Note that in the case of $Fact_{Call}$, an explicit $check_{caller}$ might not be present; instead, it may involve an implicit `owner` check for `internal` functions. Further details can be found in §4.4.

Although the definition appears straightforward, the actual facts can be complex. Specifically, there are two major challenges for MINER to effectively extract facts:

C1: *$Fact_{Def}$ could manifest in various forms across the call chain.* For example, in Figure 4 where $Fact_{Def}$ of the `transfer` function are located inside its callee function `_transfer`, the fact in line 11 is initially in the form of ["GTE" (Greater Than or Equal), "fromBalance", "amount", "ERC20: transfer amount exceeds balance"]. However, this fact cannot match with the check extracted from line 4 in a contract shown in Figure 5, which is ["GTE", "balance", "value", "AnyswapV6ERC20: transfer amount exceeds balance"]. Indeed, both the fact and the check need to propagate across the call chain to obtain their other forms so that they can eventually match with each other. We will explain more in §4.2 and §4.3.

C2: *Different $Fact_{Call}$ may intertwine within a single caller function, making them hard to distinguish.* For example, suppose the caller function is $Func_{caller}\{check_a; Func_a(); check_b; Func_b();\}$, it may be reasonable to determine that $check_a$ is relevant to $Func_a$ and similarly $check_b$ for $Func_b$. However, it is challenging to ascertain whether $check_a$ is also relevant to $Func_b$. We will elaborate more on this in §4.4.

To address these challenges, we propose a novel design for MINER, as depicted in Figure 3. It first employs Slither [28] to construct a call graph of the entire OpenZeppelin library code. In this step, we pay attention to various kinds of function calls, including not only to `internal` calls, high-level API calls, and library calls, but also to low-level calls (e.g., `abi.encodeWithSignature`). Then, for each function,

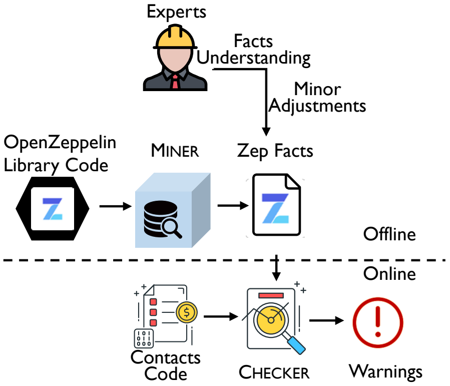


Figure 2: A workflow of ZepScope.

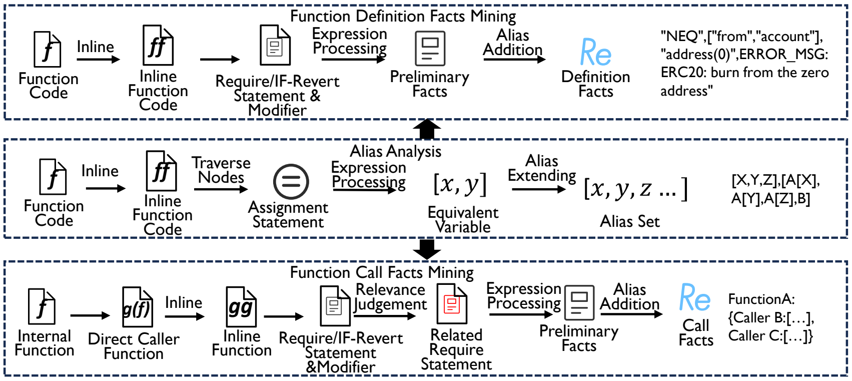


Figure 3: A detailed workflow of MINER.

```
1  function transfer(address to, uint256 amount)
       public virtual override returns (bool) {
2      address owner = _msgSender();
3      _transfer(owner, to, amount);
4      return true;
5  }
6  function _transfer(address from, address to,
       uint256 amount) internal virtual {
7      require(from != address(0), "ERC20:
           transfer from the zero address");
8      require(to != address(0), "ERC20: transfer
           to the zero address");
9      _beforeTokenTransfer(from, to, amount);
10     uint256 fromBalance = _balances[from];
11     require(fromBalance >= amount, "ERC20:
           transfer amount exceeds balance");
12     ...
13 }
```

Figure 4: The `transfer` and `_transfer` functions in the OpenZeppelin library.

```
function transfer(address to, uint256 value)
    external override returns (bool) {
    require(to != address(0) && to != address(
        this));
    uint256 balance = balanceOf[msg.sender];
    require(balance >= value, "AnyswapV6ERC20:
        transfer amount exceeds balance");
    balanceOf[msg.sender] = balance - value;
    balanceOf[to] += value;
    emit Transfer(msg.sender, to, value);
    return true;
}
```

Figure 5: An example `transfer` function from a contract.

MINER performs inter-procedural alias analysis, extracts $Fact_{Def}$ and $Fact_{Call}$, and unifies the extracted facts. We introduce these steps in detail in the following subsections.

4.2 Inter-procedural Alias Analysis

To address challenge C1, we conduct an inter-procedural alias analysis on the entire OpenZeppelin library. As depicted in Figure 3, such alias analysis can support both modules of fact extraction, enabling them to focus on the current checks without concern for their varying forms.

Based on the call graph constructed by Slither, MINER first inlines the function calls of the OpenZeppelin library. It then traverses the nodes of each function, and for each function call, it adds the caller function into the function node sets. Next, MINER traverses the assignment statements in the function nodes set and records the variables on both sides of the assignment expression as a pair of equivalent variables. For right-value (`rvalue`) expressions, it only records the stringified expressions because these expressions do not affect our equivalent variable set even if they are further processed. Subsequently, we obtain a pairwise set of equivalent variables, such as `fromBalance` and `_balances[from]` in Figure 4.

Furthermore, we extend the alias analysis from the statement level to the procedural level. For example, the variable `owner` in line 3 is aliased with the parameter `from` of the function `_transfer` according to the inter-procedural alias analysis. Our alias analysis also propagates between different sets that share the same variable, such as between the set [`fromBalance`, `_balances[from]`] and the set [`from`, `owner`]. As a result, `fromBalance` is also equivalent to `_balances[owner]`, and further equivalent to `_balances[owner]`, `_balances[_msgSender()]`, and `_balances[msg.sender]`. In this way, the `_balances[msg.sender]` part in $Fact_{Def}$ could be matched with the `balanceOf[msg.sender]` part in the extracted check of Figure 5 due to their high similarity, thus addressing challenge C1.

Note that we replace the substring (like `from` in `_balances[from]`) with the variable only in the situations where special symbols (i.e., [, (, etc.) are present before and after the substring. For other cases, we consider it a coincidental repetition of variable names, not equivalence. Additionally, to distinguish variables with the same name within a contract, we also employ the static single assignment (SSA) name and the function name to differentiate them. Finally, after recursively extending until no new equivalent variables are found, we obtain the final set of equivalent variables.

4.3 Extracting Function Definition Facts

With the alias analysis provided in §4.2, extracting $Fact_{Def}$ can concentrate on individual `require`, `if-revert`, and `modifier` statements, with other fact forms being added based on the data from the alias analysis. As illustrated in Figure 3, MINER first inlines the function calls that have $Fact_{Def}$, and then traverses the nodes of the function to analyze the `require`, `if-revert`, and `modifier` statements.

Expression Processing. For the `require` statement, we recursively process the condition expression within it using the following rules: (i) If the expression is an *AND* expression, we divide it into two individual necessary checks. (ii) If the expression is an *OR* expression, we divide it into two individual equivalent checks. (iii) If the expression is a binary expression, we traverse the expression in pre-order and record the stringified node. (iv) If the expression is a function call statement, we record the stringified node and the return value of the callee function if the return value is a global variable, a state variable, or a constant. (v) In other situations, we record the stringified node and continue to process its subexpression. Additionally, we capture the error message of the `require` statement as a part of the check facts, since some error messages can assist us in determining the type of the check. Subsequently, we obtain the facts of the require statement, such as the aforementioned ["GTE", "fromBalance", "amount", "ERC20: transfer amount exceeds balance"] for the statement line 11 in Figure 4.

For the `if-revert` statement, we first find its corresponding `if` expression. Then, we use the same rules above in the processing of `require` statements to obtain a part of the check facts. Note that we need to judge the control flow in the `if-revert` statement. Specifically, if the `revert` statement occurs in the `if` or `else if` condition, we need to reverse the logic of the extracted expression due to the opposite logic with the `require` statement. Otherwise, i.e., the `revert` statement occurs in the `else` condition, we keep the original extracted check facts. In addition, because there is no error message in the `if-revert` statements, we use the stringified node of the `revert` statement to fill the error message.

For the `modifier` statement, we address not only the `require` statement within the `modifier` statements but also record the stringified node of the modifier and its parameters. Finally, we obtain a primitive check facts list for each function in the form of $[s_1, s_2, s_3, ..., s_n]$, where s_i may be a string or a list with all equivalent elements within it.

Alias Addition. For all extracted facts, we extend the facts list using the aliases obtained in §4.2. Specifically, if a variable in the facts has equivalent variables, or its substring has equivalent variables, we add the equivalent variables to the facts list using the method mentioned earlier in §4.2: directly extend the facts list and replace the substring with the equivalent variables, then extend the facts list. In addition, we further associate the parameter with its type and the order of appearance by extending the facts list with these parameter information. Finally, we obtain the final check facts extracted from the `require` and `modifier` statements.

4.4 Extracting Function Call Facts

To extract $Fact_{Call}$, we restrict $Func_{target}$ to be the `internal` functions defined by OpenZeppelin, as the `public` and `external` functions can already be called by anyone, so the definition facts we extracted can already ensure the safety of those function calls. For example, the `internal` OpenZeppelin function `_transferOwnership` is designed to be called only within the contract, which implies that the caller is the owner of the contract. When such a function is called by a `public` function without any limitations, it suggests that anyone can call the `internal` function, which may lead to unpredictable consequences. Therefore, we need to extract the function call facts to avoid such situations.

Differing from the extraction of function definition check facts, initially, we need to identify all `internal` functions as our target functions. Then, we reverse the call graph constructed earlier and conduct a depth-first search from each target function to find the first `public` or `external` caller function of the target function. For each caller function, we regard it as a *scenario* and extract its call facts, respectively. As a result, each internal function may have $Fact_{Call}$ from multiple scenarios, i.e., different caller functions.

As shown in Figure 3, we first inline the caller function and then traverse the nodes of the caller function to find the `require` and `modifier` statements. Similar to the extraction of the definition check in §4.3, we then recursively process the expressions in the `require` and `modifier` statements using the same rules. After that, we add the aliases of the variables in the call check facts. Eventually, we obtain the final check facts extracted from the caller function of the `internal` function. It is worth noting that we also record the empty check facts, as it indicates that the callee function does not require any checks in the scenario of the caller function.

Relevance Judgement. A notable challenge here is Challenge C2 mentioned in §4.1, as there may be multiple callee functions in a caller. To distinguish the different checks associated with different callees, we apply a relevance judgment on the statement. For each statement, if it appears between two function calls, we consider it relevant to the latter callee. When there is no function call before the statement, we consider it relevant to its closest callee. Additionally, we also consider the variable that occurs in the statement. If the variables in the statements are also related to the function call,

```
1  function flashLoan(
2      IERC3156FlashBorrower receiver,
3      address token,
4      uint256 amount,
5      bytes calldata data
6  ) public virtual override returns (bool) {
7      require(amount <= maxFlashLoan(token), "
           ERC20FlashMint: amount exceeds
           maxFlashLoan");
8      uint256 fee = flashFee(token, amount);
9      _mint(address(receiver), amount);
10     require(
11         receiver.onFlashLoan(msg.sender, token,
               amount, fee, data) ==
             _RETURN_VALUE,
12         "ERC20FlashMint: invalid return value"
13     );
14     address flashFeeReceiver =
           _flashFeeReceiver();
15     _spendAllowance(address(receiver), address(
           this), amount + fee);
16     ...
17 }
```

Figure 6: The `flashLoan` function in OpenZeppelin.

we consider them relevant. Specifically, we set the variable in the statement as the source, and the variable in the function call as the sink. If there is a flow between the source and the sink, we consider them relevant.

For example, as shown in Figure 6, the function `flashLoan` is implemented by OpenZeppelin to provide a way to borrow tokens from the contract. In the function, the `_mint` function is called to mint the tokens to the borrower. However, there are some other function calls in `flashLoan`, i.e., `_flashFeeReceiver`, `_spendAllowance`, etc. Thus, we need to judge the relevance of the statement. As described above, we consider the `require` statement in line 7 to be relevant to the `_mint` function call in line 9 because `_mint` is the first called function in `flashLoan`. As for the `require` statement in line 10, due to the variables `receiver` and `amount` being relevant to the function call (i.e., function `_mint` has the parameters `receiver` and `amount`), we consider the `require` statement in line 10 to be relevant to the `_mint` function call in line 9. Therefore, the `_mint` function has two checks in the scenario of `flashLoan`.

4.5 Unifying Facts and Minor Adjustments

To enhance the accuracy and conciseness of the facts, we further unify them as some of the check facts extracted from OpenZeppelin are repeated and unnecessary.

Firstly, we eliminate repeated checks within a function. Specifically, some checks may recur because they call other functions in the definition, where both the caller and callee have check facts. Secondly, for variables associated with `msg.sender` or other constants in the equivalent variables traced, we can already determine whether the check is satisfied. For example, for the check where the variable `owner` cannot be `address(0)`, we can trace that `owner` is equivalent to `msg.sender`. Since `msg.sender` is a constant, we can already determine that the check is satisfied. Hence, we remove this check from the extracted facts. Thirdly, we make minor adjustments to the extracted facts to enhance their accuracy and conciseness. Specifically, we remove some function call facts that are redundant for the function, e.g., we extract some call facts for the function `_msgSender()`, which do not require any checks before calling. Following all these steps, we obtain the final unified facts of OpenZeppelin checks.

4.6 Understanding Facts as A Taxonomy

Having introduced the methodology of MINER from §4.1 to §4.5, we now present the results obtained from applying MINER to analyze the entire OpenZeppelin library. These results are organized as a taxonomy of OpenZeppelin checks, which, to the best of our knowledge, is a first-of-its-kind finding. Moreover, as mentioned in §4.4, $Fact_{Call}$ may yield different facts for different caller functions, i.e., *scenarios*. Thus, we also present a scenario analysis of the extracted $Fact_{Call}$.

To obtain this taxonomy, we manually review the 1,435 facts extracted by MINER. They can be divided into four major categories: *Address Compliance Assurance*, *Access Control*, *Overflow/Underflow Check*, and *Timestamp or State Check*.

① **Address Compliance Assurance.** Address compliance is a frequent check in OpenZeppelin code to ensure transaction success. It comprises four checks: `address(0)` check to prevent transfers to `address(0)`, address existence check to verify address presence in the contract, logic setup check to ensure correct logic implementation in the target address, and contract address check to confirm if the address is a contract address. These checks primarily appear in function definition facts.

② **Access Control.** Access control is another common check in OpenZeppelin functions to verify role permissions, essential for preventing economic loss in transactions. For example, the `transferOwnership` function in the `Ownable` contract verifies if the caller is the contract owner, with a simplified check through an `Ownable` modifier. OpenZeppelin library has six types of access control checks: *onlyOwner*, *onlyGovernance*, *onlyProxy*, *onlyCrossChainSender*, *onlyRoleOrOpenRole*, and *onlyRole for admin roles*. Each check verifies specific roles or permissions, like *onlyGovernance* checking if the caller is the contract's governance, and *onlyProxy* checking if the caller is the contract's proxy.

③ **Overflow/Underflow Check.** The next category comprises *overflow/underflow check*. These checks aim to prevent incorrect amounts within smart contracts. In the OpenZeppelin library, there are three types of *overflow/underflow checks*, namely, *maxtype check*, *balance check*, and *allowance check*. The maxtype check ensures that the amount does not exceed or fall below the maximum or minimum values defined for the variable type. Balance check verifies whether there is sufficient balance to carry out transactions. Allowance check ensures that the sender has the necessary allowance to execute transactions. The overflow/underflow checks are more common in the function definition facts.

④ **Timestamp or State Check.** The final type of check is the *timestamp or state check*. This type of check is often used in time-related scenarios, such as deadlines, proposal operations, transaction cancellations or pauses, and etc. These checks are often strictly dictated by the current timestamp or state. For example, to execute an `unpause` operation, the contract must be in a paused state. The timestamp or state checks are more common in the function definition facts.

Severity Analysis. Not all missed checks could lead to serious consequences. Classifying them based on severity can provide a comprehensive overview of the OpenZeppelin facts, and different severity levels also help prioritize the checking and repair of warnings. To achieve this, we use taxonomy categorization and potential attack impact to correlate individual checks with their severity levels. Specifically, the checks in the *Access control* and *Overflow/Underflow Check* categories can generally be classified by their categories, such as an overflow check as medium-level and an access control check as high-level. However, the checks in the other two categories have diverse specific types, and their severity classification needs to consider more about their attack impact. For example, within the *Address Compliance Assurance* category, while the "zero address check" is considered low-level, the absence of a "contract address check" might render the contract unusable, potentially leading to a loss of funds, and is therefore classified as medium-level. Furthermore, checks in the *Timestamp or State Check* category need to consider their impacts with the scenarios in which they are applied. For example, the check `require(block.timestamp <= expiry, "Votes: signature expired");` is considered high-level because it is related to the validity of contract votes' signatures. Eventually, our classification results in 277 high-level, 858 medium-level, and 300 low-level facts.

Scenario Analysis. Among the four categories of checks, *access control* is the most diversified check in the OpenZeppelin library. Even for the same function, the checks may vary in different scenarios. Firstly, for functions with relationship constraints, such as `transferOwnership`, a permission check is mandatory because these functions are designed to be called by the contract's owner and are crucial to the entire contract. Secondly, for functions involving assets, due to loan mechanisms or implicit checks (i.e., transactions of assets by the holder or authorized for the holder), some scenarios do not require permission checks. For example, in the caller of the `_mint` function, if there is a public `mint` to call it, we need to check the caller's permission. However, if a `flashLoan` calls it, we do not need to check the caller's permission, as there is a repayment mechanism in `flashLoan`.

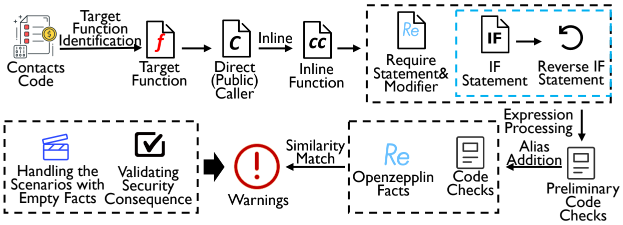


Figure 7: A detailed workflow of CHECKER.

Thirdly, some functions may use `msg.sender` as a parameter, implying an implicit check within the function. For example, in the caller of the `_transfer` function, the function `transfer` uses `msg.sender`, leading to no permission check as the function transfers the balance of `msg.sender`. Hence, we also extract caller facts when the parameter is `msg.sender` to cover these implicit checks.

5 Detecting Insecure OpenZeppelin Code in Smart Contracts

Figure 7 illustrates a high-level workflow of CHECKER, which comprises four major components: identifying target functions, extracting checks in a target function, matching the extracted code checks with facts, and finally validating the security consequences of potentially insecure code. We now elaborate on them in the following subsections.

5.1 Identifying the Target Functions

The first step is to identify the target functions within a contract, which, however, is difficult to achieve. The reason is that developers often do not directly use the OpenZeppelin code by importing the OpenZeppelin library; instead, they copy the OpenZeppelin code into their own contracts. In the process, they may rename the contract name, function name, parameter name, and even alter the code logic within the functions. Furthermore, even if they copy the original code from the OpenZeppelin repository, there are numerous versions available. Consequently, it is challenging to accurately identify the right target functions within the contract while maximizing coverage. In the current CHECKER prototype, we propose two methods to identify the target functions.

Contract-Name-Included Identification. In this method, we aim to match the target function based on the full signature of the function, which includes the contract name, function name, parameter type, and the return variable type. If the full signature of the function matches, we consider the function as the target function. Note that during this process, the variable name of the parameter does not matter; we only match the type name of the parameter. At times, a contract might inherit from its parent contract. Therefore, we further attempt to match the target function by examining the inheritance relationship of the contract if the full signature does not match. Moreover, if the contract serves as an alternative implementation of a standard, e.g., ERC20Burnable, ERC20Mintable, and we cannot match using the original name, we attempt to match using the standard name (i.e., ERC20).

Furthermore, we conduct a refinement-based identification on the function name and function parameters. During this process, we still mandate that the contract name or the parent contract name must match with our facts. Once the contract is matched, we compare the function name with the name in our facts, disregarding case differences and the special _ symbol. If the function name matches, we compare the parameter types of the function with the parameters in our facts. We allow for variations in the order of parameters and the extension of the type name of the parameter in the parameter match, e.g., we consider it a match when the parameter types are IERC3156FlashBorrower in Figure 6 and IERC3156FlashBorrowerUpgradeable in Figure 1. This refinement-based approach can handle situations where the function name and parameter types are slightly modified.

Multi-Function-Based Identification. Given that the same function signature is used across different contracts of the OpenZeppelin code, we cannot directly match the target function solely based on the signature composed of the function name, parameter types, and return variable type. To determine the origins of such ambiguous functions, we introduce a multi-function matching method. Specifically, we first match the function signature and count the number of matches within a contract. If the count equals or exceeds a pre-configured parameter, we identify all matched functions in that contract as target functions. In this study, we set this configurable parameter to 3, implying that if a contract contains three functions with matching signatures, we consider all these matched functions as target functions. Meanwhile, to mitigate mismatches between different standard contracts, for instance, when developers introduce a new set of functions in the ERC20 contract with signatures matching those in other contracts, we apply a filter to the matched functions of these standard contracts. Lastly, we execute the aforementioned refinement-based match on the function name and function parameters to enhance the multi-function match further.

5.2 Extracting Checks within Target Functions

After identifying the target functions, our next step is to extract the code checks within them. For each target function, we obtain a set of function definition facts in any scenario and a set of function call facts across different scenarios. However, unlike the extraction process within the OpenZeppelin library, here we only need to extract the code checks in the caller function of the target function to match different scenarios, since the inlined caller function encompasses the target function. Therefore, our objective is to find the first public or external function of these target functions, serving as the entry point of the call chain containing the target function. Similar to the process described in §4.4, we construct the call graph, reverse the call graph to obtain all the public or external functions in the contract, and then inline the caller function for further extraction. The extraction of these checks deviates slightly from the extraction within the OpenZeppelin library. For each caller function, we concentrate on three types of checks: require, if-revert, and modifier checks.

Require Checks. To extract checks from the require statement, we first focus on its condition expression. We break down the expression into several independent inspections using the logical operators && or ||||. For each inspection, we perform a recursive pre-order traversal to extract the checks. At the leaf node of the expression, if the node type is a function call and the function's return is a state variable, a global variable, or a constant, we record both the stringified node and the return value. Otherwise, we only record the stringified node. After inspecting the condition expression, if the statement has an error message parameter, we record the error message at the end of the check. If not, we record "*no error message*". For each divided inspection, we use the same error message as the parent inspection.

If-Revert Checks. Besides require statement, another insecure situation is the If-Revert statement. In this statement, developers might use an if statement to evaluate a condition and a revert statement to throw an error message if the condition is met. In this scenario, we also need to extract checks from the if statement. The extraction process for if-revert statements is similar to that for require statements. We also divide the condition expression using the logical operators && or |||| into several independent inspections and perform a recursive pre-order traversal on them to extract the checks. However, the logic in if-revert statements is opposite to that in require statements, so we reverse the logic in the extracted checks. Lastly, as the if statement does not have an error message parameter, we only record the stringified node of the revert statement.

Modifier Checks. Regarding modifiers, since we have already inlined the caller function and extracted require and if-revert statements from it, there is no need to specifically extract checks from the modifiers. However, to avoid false positives in CHECKER due to some transformations of the checks extracted in the above statements, we extract the modifier name as a check and match it in the subsequent step.

For all checks extracted from these three statements, we also need to conduct an alias analysis and augment them as described in §4.2. Eventually, we obtain a set of code checks in the caller function of the target function, formatted similarly to the facts mined in §4, ready to be matched subsequently.

5.3 Matching the Code Checks with Facts

We now proceed to compare the extracted code checks with the facts mined from the OpenZeppelin library. For each caller of the target function, we compare the extracted code checks with the facts in the function definition and each scenario of function call. As shown in Figure 7, our matching process comprises three steps: matching error messages, matching the checks, and handling the scenarios with empty facts.

5.3.1 Matching Error Messages

Error messages, like the aforementioned "ERC20: transfer amount exceeds balance," signify the content of the checks. These messages can help distinguish different types of checks. Therefore, we first try to match the extracted error message with the facts mined from OpenZeppelin. Given that the error message might include the contract name, e.g., `ERC20:`, it is necessary to replace the contract name in the error message prior to the symbol ":". After that, we conduct a similarity match as follows. We first tokenize the messages to isolate individual words, map each word to its corresponding vector in the Word2Vec [42] embedding space, and compute the average of these vectors to obtain a sentence-level vector. Finally, we calculate the Word Mover's Distance (WMD) [37] similarity between the sentence vectors, given its effectiveness in measuring the semantic similarity of short sentences [51].

Formally, Let M_1 and M_2 be two messages to be compared (e.g., one from the error and the other from the fact). Each messages M_i (where $i \in \{1,2\}$) consists of n_i words, such that $M_i = \{w_{i1}, w_{i2}, ..., w_{in_i}\}$. Each word w_{ij} (where $j \in \{1, ..., n_i\}$) in messages M_i is mapped to a vector $v_{ij} \in \mathbb{R}^d$ using the Word2Vec model, where d is the dimensionality of the Word2Vec embedding space. Thus, the messages M_i is represented by a set of vectors $\{v_{i1}, v_{i2}, ..., v_{in_i}\}$. The vector representation of the entire messages M_i, denoted as V_{M_i}, is calculated by averaging the vectors of all words in the sentence, i.e., $V_{M_i} = \frac{1}{n_i}\sum_{j=1}^{n_i} v_{ij}$. The semantic similarity between messages M_1 and M_2 is quantified using the WMD similarity between their vector representations V_{M_1} and V_{M_2}. The WMD similarity, denoted as $\text{sim}(M_1, M_2)$, is computed as follows:

$$\text{sim}(M_1, M_2) = 1 - \min_{T \geq 0} \sum_{i,j=1}^{n} T_{ij} \cdot c(i, j) \qquad (1)$$

T is a transport matrix, where T_{ij} denotes the amount of "mass" being transported from the ith word in M_1 to the jth word in M_2. Note that $c(i, j)$ is the cost function, which represents the distance between the ith word and the jth word, calculated as the Euclidean distance between their vector embeddings.

Following this, we set a threshold T_{error} to filter the matched error message. Since error messages serve only as a pre-filtering step, we must be conservative and prioritize using a higher threshold value for T_{error}. According to our measurement of two major threshold values (0.8 and 0.9) in Appendix C.1, we eventually chose 0.9 as the value for T_{error} in this paper. In addition, we also conduct an ablation study in Appendix D to compare the effectiveness of two similarity matching strategies described in §5.3.1 and §5.3.2. If the error message matches in such a conservative way, we directly consider the check to be matched. For cases not matched, we conduct further matching without the error messages.

5.3.2 Matching the Checks

Next, we try to match the checks without error messages. As previously mentioned, the extracted checks and facts are structured in the format $[s_1, s_2, s_3, ..., s_n]$, where s_i may be a string or a list with all elements being equivalent within it. Therefore, we first match each $s_{i_{extracted}}$ and $s_{i_{fact}}$ in order. If both $s_{i_{extracted}}$ and $s_{i_{fact}}$ are strings, we utilize the edit distance to calculate the similarity between s_i and the fact. Specifically, this similarity is computed by Equation 2:

$$\text{Similarity} = 1 - \frac{edit(s_{i_{extracted}}, s_{i_{fact}})}{\min\{\text{length}(s_{i_{extracted}}), \text{length}(s_{i_{fact}})\}} \qquad (2)$$

where $edit(s_{i_{extracted}}, s_{i_{fact}})$ denotes the edit distance between $s_{i_{extracted}}$ and $s_{i_{fact}}$.

If one is a substring of the other, the similarity is 1. If a list is present in either $s_{i_{extracted}}$ and $s_{i_{fact}}$, the similarity is the maximum similarity between the elements in the list and the string. For situations where both $s_{i_{extracted}}$ and $s_{i_{fact}}$ are lists, the similarity is the maximum similarity of the elements.

Subsequently, we set a threshold T_s to filter the matched s_i. If the similarity of s_i exceeds the threshold, we consider the element to be matched. Otherwise, we attempt to match the next element. In Appendix §C.1, we measure how different threshold values of T_s could impact the final results and eventually chose 0.6 as the value for T_s in this paper.

With each element s_i's similarity determined, we can calculate the similarity of the entire check. Here, we choose not to use typical NLP-based similarity methods because they might overlook the structural information embedded in each fact check. For example, for an original fact `require(amount < fromBalance)`, a typical NLP-based matching might consider `require(amount > fromBalance)` more similar than `require(oldBalance > newAmount)`. Therefore, we propose *structured similarity matching*, which is a lightweight yet effective approach to our problem. Specifically, it structurizes each individual matching unit (i.e., the element s_i above), including symbols like <, measures each similar unit based on Equation 2, and conducts majority voting with similar units in comparison to the total number of units. For symbols, we also perform an equivalence check in §5.4.

For the threshold used by this structured similarity matching, we design a dynamic factor F to effectively conduct majority voting across different numbers of total units:

$$F = \lceil \frac{\text{unit\_number}(S)}{2} \rceil + 1 \qquad (3)$$

where S is the extracted fact. If the number of similar elements in the checks equals or exceeds the dynamic factor, we consider the check to be matched. For function call facts that have more than one scenario, we consider the check to be matched if the number of similar elements satisfies the dynamic factor in any given scenario. In Appendix C.2, we also compared F with different fixed factors and found that the dynamic factor outperforms them.

5.3.3 Handling the Scenarios with Empty Facts

Empty checks exist within the function call facts across different scenarios, indicating that OpenZeppelin omitted checks in some scenarios of the function call. However, these empty facts might result in missing warnings during the matching process above, as any extracted checks can fulfill these facts. Therefore, we further handle the scenarios with empty facts. Specifically, we divide the contract name and the function name into words and count repeated words to represent the similarity between the detected scenario and the facts scenario. Following this, we employ Equation 3 to calculate and set the similarity threshold. Should the similarity of the scenario surpass the threshold, we consider the scenario as matched and believe that this scenario necessitates no checks.

5.4 Validating Security Consequences

The missed code checks that are not matched in the last step are only potentially insecure OpenZeppelin code because they may have other equivalent checks either explicitly or implicitly. Therefore, CHECKER further performs this validation step to assess the security consequences. In this study, we focus on four types of validation, namely, equivalent overflow protection, equivalent permissions, extra `msg.value` checks, and logical equivalence conditions,.

Equivalent Overflow Protection. In OpenZeppelin code, developers often use explicit judgments, such as `require(amount < fromBalance)`, to avoid overflow and underflow issues. However, developers may also use implicit judgments, for instance `fromBalance.sub(amount)`, where the `sub` function reverts when `fromBalance` is less than `amount`. Moreover, with the improvements in the Solidity compiler, specifically from version 0.8.0 onwards, Solidity now includes built-in checks to prevent integer overflows or underflows, eliminating the need for developers to manually add overflow checks for every arithmetic operation. Therefore, it is necessary to validate the equivalent overflow protection in CHECKER. Specifically, we first ascertain the version of Solidity and disable the overflow checks if the version is 0.8.0 or later. Notably, for facts with checks like `amount < fromAllowance`, which not only serve as checks for overflow and underflow but also as checks for the amount of allowance, we still examine the `allowance`-related aspects in the function to ensure the caller has sufficient allowance to invoke the function. Next, we look for any form of explicit judgment for overflow checks. If such a check is missing, we then check for any form of implicit judgment, i.e., using SafeMath to avoid overflow and underflow. If the check is still missing after these validations, only then do we issue a warning.

Equivalent Permissions. As described in §4.6, permission checks vary across different functions. For example, the `safetransferFrom` function in the ERC721 contract requires the `_isApprovedOrOwner` function, while the `mint` function in the ERC20 contract necessitates the sender to possess the `minter` role. Additionally, some developers may utilize the modifier `onlyOwner` to achieve this. It is thus important to validate the equivalent permission in CHECKER. Based on the understanding derived from the facts extracted from OpenZeppelin, we rely on the following hierarchy of authority. Primarily, the modifier `onlyOwner` or the `require` judgement `msg.sender == owner` represents the strictest level of permissions. Following this, the modifiers `onlyGovernance`, `onlyAdmin`, and `onlyAuthority`, alongside their corresponding `require` judgements, as well as the functions `_isApprovedOrOwner` and `isApprovedForAll` constitute a secondary tier of strict permissions. Lastly, the requirement for roles of other minor authorities denotes the most lenient permissions. Within CHECKER, we determine whether the requirements are fulfilled based on our categorized permissions. Moreover, we have extracted `msg.sender` in caller parameters as a fact, which also represents a form of permission check. We deduce that the check is satisfied if the caller has executed the aforementioned permission checks.

Extra msg.value Checks. Furthermore, some contracts may incorporate an extra `msg.value` check within a function to prevent unrestricted calls to the function by everyone. For example, developers might define a `buy` function to facilitate token exchanges with users. This function requires that `msg.value` exceeds a certain value, following which some tokens are minted to complete the exchange. In this function, the caller of the `_mint` function does not require a check on the permission of `msg.sender`, as the `msg.value` check has already restricted the calls. Therefore, to accommodate this scenario, we implement an extra `msg.value` check in CHECKER. Specifically, we first conduct the check based on the extracted facts. Then, if a permission check is found to be missing, we carry out a `msg.value` check in the caller function. Eventually, we will flag the case if a `msg.value` check is present, or report a missing permission check otherwise.

Logical Equivalence Conditions. The extracted facts for binary expressions are typically formatted as [EQ, variableA, variableB, Error Message], indicating that variable A must equal to variable B. However, developers may use the logically equivalent condition, e.g., `variableB == variableA`, which may be extracted as [EQ, variableB, variableA, Error Message]. Likewise, for the condition where variable A is greater than or equal to variable B, developers could alternatively express this as variable B being less than or equal to variable A. Therefore, it is important to validate the logical equivalence condition in CHECKER. Specifically, we first check whether the extracted facts are in the original form if the facts include relational operators. If the check is missing, we then verify the equivalent form of the facts, e.g., for *greater than*, we substitute with *less than*. If the check is still missing following these steps, only then do we issue a warning.

6 Evaluation

In this section, we conduct a thorough evaluation of ZepScope's effectiveness and usefulness by addressing the following research questions (RQs):

RQ1: How does the effectiveness of ZepScope compare to that of other state-of-the-art (SOTA) static analysis tools?

RQ2: How accurate and efficient is ZepScope in analyzing a large set of real-world smart contracts?

RQ3: Can ZepScope identify notable security findings?

RQ4: Concerning OpenZeppelin checks in smart contracts, what are the primary differences across three major Ethereum-compatible chains?

Experimental Setup. As mentioned in §3, ZepScope consists of two components: MINER and CHECKER. We implement each component within the framework of Slither [28], a fundamental framework for tools developed in many academic papers [21,41,52]. ZepScope complements Slither by proposing (i) a mining approach for extracting OpenZeppelin function checks in §4 and (ii) a set of customized analysis techniques in §5 for detecting insecure OpenZeppelin code in smart contracts. To reduce the compilation time of the contracts, we separate and optimize the compilation part of Slither (i.e., crytic-compile [4]) with automatic compiler version selection. This way, we can compile the contracts first and then analyze them with CHECKER. All the experiments were conducted on a server equipped with 80 vCPUs (Intel(R) Xeon(R) Gold 6248 CPU @ 2.50GHz × 2) and 188G of RAMs.

6.1 RQ1: Comparison with the SOTA Tools

In this RQ, we aim to benchmark ZepScope against some state-of-the-art (SOTA) static analysis tools, particularly those that claim to detect *Access Control* and *Overflow/Underflow* vulnerabilities, which are two common types of issues that insecure OpenZeppelin code can cause; see details in in §4.6. Specifically, we want to compare the effectiveness of fact-based detection employed by ZepScope and pattern-based detection used by other SOTA tools in identifying these critical issues related to OpenZeppelin checks.

Following this objective, we compare ZepScope with three SOTA tools: Slither [28], AChecker [31], and SoMo [27]. Specifically, Slither serves not only as a supporting framework for numerous smart contract tools [53] but also as a standalone tool that has integrated detection rules for many issues including access control and overflow. AChecker and SoMo are two recent SOTA analysis tools focused on access control problems, with AChecker detecting general access control issues through critical instruction-driven taint analysis, and SoMo targeting specific access control issues related to bypassable modifiers. Both Slither and AChecker can be run on our testbed as they are open-source. However, since SoMo is not open-source and is integrated into an industry scanning platform called MetaScan [13], we utilize MetaScan's scanning results, which also include overflow detection.

Table 1: Benchmarking ZepScope against three SOTA tools on 51 real-world security bugs with insecure OpenZeppelin code.

Tool	TP	FP	FN	# Failed
Slither	8	32	43	0
AChecker	0	0	43	8
SoMo (via MetaScan)	8	22	43	0
ZepScope	41	0	10	0

To evaluate these tools in the context of our problem, we manually collected 51 real-world security bugs caused by insecure OpenZeppelin code. These bugs were sourced from security incidents reported on DeFiHackLabs [5], Twitter [18, 19], SmartBugs Curated datasets [26], and audit reports from Code4rena [3], Sherlock [17], and Ethereum Commonwealth [20]. Among these 51 ground-truth issues, 25 pertain to access control, five are related to arithmetic overflow, one belongs to front running, and the remaining 20 are caused by insecure OpenZeppelin checks. We then utilized these ground-truth vulnerabilities as the benchmark dataset. Note that we tallied the results at the function level, considering a vulnerability as detected if the function is identified by the tool and the categories of the vulnerability match.

Table 1 shows the benchmarking results, including the number of true positives (TP), false positives (FP), false negatives (FN), and failed cases. Notably, out of 51 ground-truth cases, ZepScope detected 41. The missed issues are due to: (i) For calls formatted as member access (e.g., `IERC20.SafeTransfer`), we were unable to find the definition call chain, which could be defined either within or outside the contract. We chose to ignore this call format to avoid additional false alarms. (ii) Vulnerable logic occurs in one of the data flows within the function, while the function has met the criteria in our check facts, leading to false negatives. (iii) Function matching failed because some of the ground-truth function signatures have changed significantly, which caused us to miss this function during the matching process.

In contrast, the other tools yielded underperforming results: Slither detected only 8 out of 51, with 32 false positives;

Table 2: The evaluation data of ZepScope on 35,882 contracts.

Chain	# Contacts	# Failed	# Functions	# Warnings	Sampled Accuracy
Ethereum	13,984	150	911,309	16,873	84%
BSC	12,486	234	1,068,035	14,444	95%
Polygon	9,955	159	770,821	16,114	90%
Total	36,425	543	2,750,165	47,431	89.67%

AChecker detected none; and SoMo via MetaScan also reported only 8 true positives. The primary reasons for their failure are: (i) Detecting missing access control requires a detection mechanism that understands and adapts to different access control scenarios, which the existing tools failed to accomplish. (ii) The arithmetic overflow also went undetected by all the tested tools (except ZepScope), as it could only be triggered under extreme scenarios (i.e., lines 13 - 15 in the motivating example shown in Section 2.2). (iii) The rest of the issues that require AC-related domain knowledge, such as those from Ethereum Commonwealth [20], are difficult to detect with existing tools.

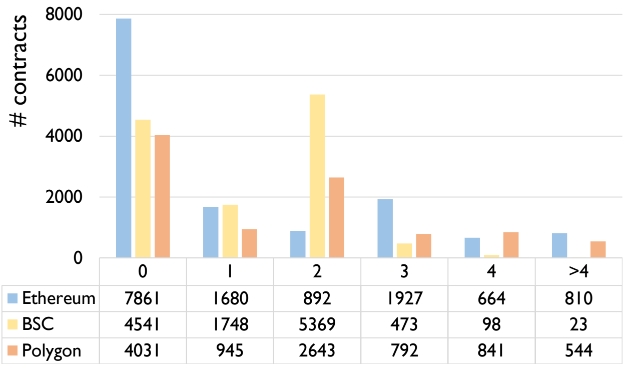
Thus, ZepScope significantly outperforms in detecting the vulnerability types that, although covered by SOTA tools, are related to OpenZeppelin checks. This superior performance is attributed to ZepScope's unique approach of understanding OpenZeppelin facts and leveraging them for detection.

6.2 RQ2: Accuracy and Performance

In this RQ, we further evaluate ZepScope's accuracy and performance on a large set of real-world smart contracts collected from three mainstream Ethereum-compatible chains: Ethereum, BSC, and Polygon.

This large-scale dataset was assembled from the top 15,000 contracts of each chain, ranked by the balances of the contracts. Note that this crawling process was conducted through the APIs of Etherscan, BscScan, and PolygonScan, which would have download limits and failures. We thus downloaded all the contracts we could and compiled them for further analysis. After excluding the sections with download and compilation failures, we obtained 13,984, 12,486, and 9,955 contracts in Ethereum, BSC, and Polygon, respectively. We analyzed these contracts using ZepScope with 60 parallel processes and set the maximum scanning time to 5 minutes each.

As shown in Table 2, ZepScope successfully analyzed 35,882 contracts while encountering failures in 543 contracts. The failures on all three chains were primarily due to timeouts. Ultimately, ZepScope reported 47,431 functions potentially containing insecure OpenZeppelin code out of the total 2,750,165 functions analyzed. This implies that ZepScope generates around one warning per contract, which is entirely manageable for further manual inspection. Furthermore, based on the severity levels of different facts as classified in §4.6, we categorized these warnings into 3,015 high-level warnings, 5,161 medium-level warnings, and 39,255 low-level warnings to facilitate a more principled analysis in §6.2.



	0	1	2	3	4	>4
Ethereum	7861	1680	892	1927	664	810
BSC	4541	1748	5369	473	98	23
Polygon	4031	945	2643	792	841	544

Figure 8: The distribution of warnings on 35,882 contracts.

Accuracy Evaluation. Due to the absence of ground truth in such a large-scale dataset, we randomly sampled 100 warnings from each chain and conducted a careful manual inspection on them. To ensure correctness, the results were reviewed by two of the authors. In case of a disagreement between the two authors, a third author was consulted. Among the total of 300 warnings inspected, we confirmed that 31 were false positives.Table 2 presents the detailed accuracy results, showing that ZepScope performed best in the BSC contracts, with an accuracy of 95%, while the accuracy for Ethereum and Polygon were 84% and 90%, respectively. The lower accuracy for Ethereum can be attributed to its contracts being more complex compared to the other chains, often featuring more intricate logic to extend the OpenZeppelin code. Overall, the accuracy of ZepScope stands at 89.67% across all the sampled contracts. Moreover, not every contract triggers a warning. We analyzed the distribution of warnings on each chain, as shown in Figure 8, and found that 45.8% of the contracts (56.8% for the contracts on Ethereum) do not require manual confirmation because they do not report any warnings.

We further categorize the reasons for the 31 false positives into two types:check without revert and check in equivalent format. For details, readers may refer to Appendix A.

Performance Evaluation. During the experiment, we also recorded the scanning time of ZepScope while analyzing contracts in the dataset. Since the contracts were pre-compiled, the compilation time is not included. In total, ZepScope took 25,734 seconds to scan the 36,425 contracts with 60 parallel processes. That said, it requires 42.39 seconds on average for ZepScope to analyze a contract without considering parallel processing. This suggests that ZepScope is quite fast, making it suitable for large-scale on-chain scanning.

6.3 RQ3: Security Findings

In this RQ, we present three notable security findings obtained from the large-scale experiment conducted in §6.2.

6.3.1 Newly Identified Vulnerabilities

We conduct a manual analysis to investigate whether ZepScope can detect OpenZeppelin code issues that belong to the definition of vulnerabilities. Specifically, we first filter the high-level warnings reported by ZepScope. We then manually review some warnings related to role checks, which could lead to significant consequences. Eventually, we identified 15 new vulnerabilities. Among them, three could directly transfer or mint tokens to gain profits, 11 could burn victims' tokens causing financial loss and indirectly benefiting attackers, and the last one could render the contract ownerless. In total, these 15 vulnerable contracts could cause a financial loss of around $439,333. Current methods mostly rely on restricted predefined patterns and critical operations, which may not cater to the diverse requirements found across different business logic scenarios, e.g., different access control requirements for burn/mint (e.g., ERC20-mint, flashloan, buy, and airdrop). In contrast, ZepScope utilizes the facts mined from the OpenZeppelin library to effectively address complex scenarios and detect these vulnerabilities.

Case Study 1. As illustrated in Figure 11 (in Appendix B), there is a vulnerable `transferOwnership` function in a contract A. Its developers set this function as a `public` function without incorporating any role check, allowing anyone to call this function to change the contract's owner. Furthermore, the contract contains functions (e.g., `setTradingStatus`) with the `onlyOwner` modifier, which could be bypassed by calling the `transferOwnership` function.

Case Study 2. As depicted in Figure 12 (in Appendix B), a vulnerable `burn` function is present in a contract B. The `burn` function is designed to be called by the contract owner or the token owner to burn the tokens. However, this `burn` function lacks any permission checks within its function definition, enabling anyone to call this function to burn others' tokens, potentially leading to significant asset loss.

Ethics. Due to the reasons outlined in teEther [36], we are unable to reach out to the developers of these vulnerable contracts. Fortunately, these contracts have not shown much transaction activity, indicating that the financial stakes have been relatively low, thus minimizing potential losses. In this paper, we only provide code snippets without disclosing the contract names to prevent potential attacks.

6.3.2 Pervasive Absence of Zero Address Checks

In the OpenZeppelin library, certain functions execute checks on the input address parameter to prevent misuse of `address(0)`. For example, as illustrated in Figure 4 within the `_transfer` function of ERC20, the parameters `from` and `to` are required not to be `address(0)` to (i) avoid unintentional permanent locking of tokens due to human errors or software glitches; (ii) differentiate the `_transfer` function from the `burn` function, clearly indicating that transferring to a zero address is erroneous behavior rather than intentional; and (iii) avoid inaccuracies in the total supply figures while also preventing extra gas fee loss. Similar checks for `address(0)` are necessary in other functions such as `mint` and `transferOwnership`.

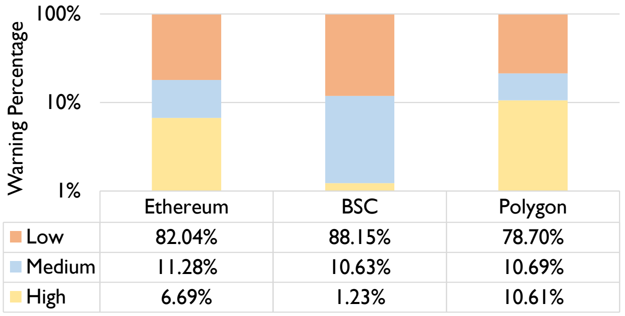
```
 1 function buy(address _refer) payable public
       returns(bool){
 2     require(_swSale && block.number <=
           saleMaxBlock,"Transaction recovery");
 3     require(msg.value >= 0.01 ether,"
           Transaction recovery");
 4     uint256 _msgValue = msg.value;
 5     uint256 _token = _msgValue.mul(salePrice);
 6     _mint(_msgSender(),_token);
 7     if(_msgSender()!=_refer&&_refer!=address(0)
           &&_balances[_refer]>0){
 8         uint referToken = _token.mul(
               _referToken).div(10000);
 9         uint referEth = _msgValue.mul(_referEth
               ).div(10000);
10         _mint(_refer,referToken);
11         address(uint160(_refer)).transfer(
               referEth);
12     }
13     return true;
14 }
```

Figure 9: The `buy` function in Shiba_Inu.

Nonetheless, ZepScope identifies numerous functions that omit the `address(0)` check. Through our analysis of ZepScope's results, we discovered that 22,448 functions lack `the address(0)` check, marking a common negligence. While the omission of an `address(0)` check cannot be directly exploited by attackers, it can pave the way for phishing attacks. For example, attackers might deceive users into sending funds to a zero address (seemingly harmless) through a counterfeit website. As tokens sent to the zero address are consequently removed from circulation, the overall token supply decreases. This induced scarcity can potentially augment the value of the remaining tokens, benefiting attackers who hold these tokens. Therefore, we recommend developers incorporate `address(0)` checks in OpenZeppelin code to mitigate phishing attacks and standardize the transaction process.

6.3.3 A Campaign of Intentionally Loosing the Checks

Besides vulnerabilities and the pervasive absence of zero address checks, we identify an interesting case where OpenZeppelin checks are intentionally loosened, yet without causing security consequences. As depicted in Figure 9, the `buy` function in a contract named Shiba_Inu invokes the `_mint` function (line 6 and 10) without any access control, which



	Ethereum	BSC	Polygon
🟧 Low	82.04%	88.15%	78.70%
🟦 Medium	11.28%	10.63%	10.69%
🟨 High	6.69%	1.23%	10.61%

Figure 10: Percentage of warnings at different levels on 3 chains.

initially violates OpenZeppelin checks. Fortunately, this function also incorporates a custom `msg.value` check in line 3, requiring the sender to transmit an ETH or BNB amount larger than 0.1 to the contract to gain the capability of minting a certain amount of the contract token. Therefore, according to our security consequence validation in §5.4, this issue only qualifies as a low-level warning, urging further comparison between the value of the sent ETH/BNB and the minted tokens. Nonetheless, this example illustrates that, at times, developers may loosen the default checks to accommodate their own logic, which should be safeguarded by their own checks.

Indeed, the scenario presented in Figure 9 is not an isolated case. We discovered that BSC alone hosts 255 `buy` functions exhibiting similar logic. They are all crafted for extensive promotion of the tokens reliant on this function, suggesting that this practice is part of a campaign or a common approach for such airdrop-like token sales. Moreover, concerning the extra `msg.value` check, ZepScope identifies a total of 8,061 functions containing it, indicating that the function can only be invoked with `msg.value` greater than 0. This reveals that such general checks, extending beyond OpenZeppelin's default checks, are prevalent in smart contracts, further underscoring the importance of our security consequence validation as discussed in §5.4.

6.4 RQ4: Cross-Chain Result Comparison

In this RQ, we explore the differences in OpenZeppelin security checks across three chains. To this end, we utilize the results presented in §6.2 and conduct a cross-chain comparison. We calculate the proportion of warnings at different levels on three chains and depict the result in Figure 10.

Firstly, we notice that BSC outpaces the other two chains with the highest proportion of low-level warnings at 88.15%. This surge in BSC's low-level warnings may be attributed to the frequent omission of `address(0)` checks, which fall under the low-level category. Secondly, Ethereum presents a warning distribution with 82.04% at low, 11.28% at medium, and 6.69% at high levels. A significant factor to Ethereum's warning landscape is the presence of numerous legacy contracts. For example, we found that 1,473 out of 13,984 contracts on Ethereum were deployed before 2019, and 1,408 contracts were deployed before 2018. These contracts might have been deployed before the emergence of OpenZeppelin libraries, thereby resulting in worse security protection. Thirdly, Polygon reports 78.70%, 10.69%, and 10.61% for low, median, and high warning levels, respectively. A notable observation here is the prevalent use of the `functionCallValue` function in Polygon. While OpenZeppelin libraries have checks for this function in place, Polygon seems to bypass them, potentially influencing its warning distribution.

7 Related Work

In this section, we briefly review prior work in the domain of smart contract analysis, focusing on static analysis-based vulnerability detection and code clone detection.

General static analysis tools, such as Slither [28], Vandal [23], Ethainter [22], Zeus [33], Securify [50], and 4naly3er [45], have been developed to identify a range of vulnerabilities in smart contracts. For detecting vulnerabilities related to variable boundary values, symbolic execution tools such as Manticore [43], Mythril [14], Halmos [10], and Pyrometer [16] have emerged. These tools typically facilitate the analysis of variable boundaries, either directly from source code or from the compiled Solidity bytecode. There exist specialized static detectors tailored for detecting certain classes of vulnerabilities, including reentrancy [49], arithmetic overflow [48], state inconsistency [21], and access control issues [27,31,40]. Formal verification tools, like Verx [44] and Verismart [46], have been developed to ascertain if a given smart contract aligns with its specified requirements.

Recent investigations into the ecosystem's code practices reveal that code cloning is prevalent in smart contracts [24,29,35,38,39,47]. Notably, Khan et al. [34] discerned that a staggering 79.2% of code on the Ethereum platform comprises clones. Based on this finding, Gao et al. [30] and He et al. [32] employed code clone techniques to uncover vulnerabilities within the smart contract ecosystem. While these studies offer valuable insights, they predominantly concentrate on type-2 and type-3 clones, thereby overlooking subtle, semantic-level changes in code that could hold critical importance. Our work, based on the code facts extracted by MINER, is the first to systematically study insecure OpenZeppelin code used in real-world contracts.

8 Conclusion

In this paper, we presented the first systematic study of insecure OpenZeppelin code used in real-world smart contracts. To achieve this, we proposed ZepScope, a novel tool for extracting code check facts from OpenZeppelin and subsequently identifying fact violations in real-world contracts. In a benchmark experiment with 51 ground-truth security bugs, ZepScope significantly outperformed in detecting the vulnerability types that, although covered by SOTA tools, are related to OpenZeppelin checks. When applied to 35,882 leading contracts across three mainstream blockchains, ZepScope demonstrated an accuracy of 89.67% and yielded insightful results. Future work includes improving the accuracy of equivalence checks and target function identification. We also plan to extend our methodology to other smart contract libraries.

Acknowledgements

We thank the anonymous shepherd and reviewers for their constructive feedback. This research is supported by the National Research Foundation, Singapore, and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-GC-2023-008), 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 National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

[1] An attacker can cause an overflow in the flashloan function. https://github.com/code-423n4/2021-05-nftx-findings/issues/75, Oct 2023.

[2] Blocksec attack analysis. https://twitter.com/BlockSecTeam/status/1692533280971936059, Oct 2023.

[3] Code4rena audit reports. https://code4rena.com/reports, Oct 2023.

[4] crytic-compile. https://github.com/crytic/crytic-compile, Oct 2023.

[5] Defi hacks. https://github.com/SunWeb3Sec/DeFiHackLabs, Oct 2023.

[6] Erc-20. https://ethereum.org/en/developers/docs/standards/tokens/erc-20/, Oct 2023.

[7] Erc-721. https://ethereum.org/en/developers/docs/standards/tokens/erc-20/, Oct 2023.

[8] Erc-777. https://ethereum.org/en/developers/docs/standards/tokens/erc-777/, Oct 2023.

[9] Ethereum whitepaper. https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf, Oct 2023.

[10] Halmos. https://github.com/a16z/halmos, Oct 2023.

[11] Kev0. https://bscscan.io/address/0x6835E6539bBD0975a324331D4EBd26DC16031F68, Oct 2023.

[12] Lifeforms. https://etherscan.io/address/0x61f68f7db9dee422991f410a8d863f538181d7c1#code, Oct 2023.

[13] Metascan. https://metatrust.io/metascan, Oct 2023.

[14] Mythril. https://github.com/Consensys/mythril, Oct 2023.

[15] Openzeppelin. https://www.openzeppelin.com, Oct 2023.

[16] Pyrometer. https://github.com/nascentxyz/pyrometer, Oct 2023.

[17] Sherlock audit. https://github.com/sherlock-audit, Oct 2023.

[18] Tittwer report 1 in peckshield. https://twitter.com/peckshield/status/1640847953098334208, Oct 2023.

[19] Tittwer report 2 in peckshield. https://twitter.com/peckshield/status/1654667621139349505, Oct 2023.

[20] Callisto smart-contract auditing department. https://github.com/EthereumCommonwealth/Auditing, May 2024.

[21] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *Proc. IEEE Symposium on Security and Privacy*, 2022.

[22] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proc. ACM PLDI*, 2020.

[23] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*, 2018.

[24] Xiangping Chen, Peiyong Liao, Yixin Zhang, Yuan Huang, and Zibin Zheng. Understanding code reuse in smart contracts. In *Proc. IEEE SANER*, 2021.

[25] Dipanjan Das, Priyanka Bose, Nicola Ruaro, Christopher Kruegel, and Giovanni Vigna. Understanding security issues in the NFT ecosystem. In *Proc. ACM CCS*, 2022.

[26] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *Proc. ACM/IEEE ICSE*, 2020.

[27] Yuzhou Fang, Daoyuan Wu, Xiao Yi, Shuai Wang, Yufan Chen, Mengjie Chen, Yang Liu, and Lingxiao Jiang. Beyond "protected" and "private": An empirical security analysis of custom function modifiers in smart contracts. In *Proc. ACM ISSTA*, 2023.

[28] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE, 2019.

[29] Zhipeng Gao. When deep learning meets smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1400–1402, 2020.

[30] Zhipeng Gao, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. Checking smart contracts with structural code embedding. *IEEE Transactions on Software Engineering*, 2020.

[31] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. AChecker: Statically detecting smart contract access control vulnerabilities. In *Proc. IEEE/ACM ICSE*, 2023.

[32] Ningyu He, Lei Wu, Haoyu Wang, Yao Guo, and Xuxian Jiang. Characterizing code clones in the Ethereum smart contract ecosystem. In *Proc. Springer FC*, 2020.

[33] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: Analyzing safety of smart contracts. In *Proc. ISOC NDSS*, 2018.

[34] Faizan Khan, Istvan David, Daniel Varro, and Shane McIntosh. Code cloning in smart contracts on the Ethereum platform: An extended replication study. *IEEE Transactions on Software Engineering*, 49(4):2006–2019, 2022.

[35] Masanari Kondo, Gustavo A Oliva, Zhen Ming Jiang, Ahmed E Hassan, and Osamu Mizuno. Code cloning in smart contracts: a case study on verified contracts from the Ethereum blockchain platform. *Empirical Software Engineering*, 25:4617–4675, 2020.

[36] Johannes Krupp and Christian Rossow. teEther: Gnawing at ethereum to automatically exploit smart contracts. In *Proc. USENIX Security Symposium*, 2018.

[37] Matt J. Kusner, Yu Sun, Nicholas I. Kolkin, and Kilian Q. Weinberger. From word embeddings to document distances. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, 2015.

[38] Han Liu, Zhiqiang Yang, Yu Jiang, Wenqi Zhao, and Jiaguang Sun. Enabling clone detection for ethereum via smart contract birthmarks. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 105–115. IEEE, 2019.

[39] Han Liu, Zhiqiang Yang, Chao Liu, Yu Jiang, Wenqi Zhao, and Jiaguang Sun. Eclone: Detect semantic clones in Ethereum via symbolic transaction sketch. In *Proc. ACM FSE*, 2018.

[40] Ye Liu, Yi Li, Shang-Wei Lin, and Cyrille Artho. Finding permission bugs in smart contracts with role mining. In *Proc. ACM ISSTA*, 2022.

[41] Benjamin Mariano, Yanju Chen, Yu Feng, Shuvendu K Lahiri, and Isil Dillig. Demystifying loops in smart contracts. In *Proc. IEEE/ACM ASE*, 2020.

[42] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[43] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *Proc. ACM/IEEE ASE*, 2019.

[44] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *Proc. IEEE Symposium on Security and Privacy*, 2020.

[45] Picodes. 4naly3er. https://github.com/Picodes/4naly3er, Oct 2023.

[46] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. VeriSmart: A highly precise safety verifier for Ethereum smart contracts. In *Proc. IEEE Symposium on Security and Privacy*, 2020.

[47] Kairan Sun, Zhengzi Xu, Chengwei Liu, Kaixuan Li, and Yang Liu. Demystifying the composition and code reuse in Solidity smart contracts. In *Proc. ACM FSE*, 2023.

[48] Bryan Tan, Benjamin Mariano, Shuvendu K. Lahiri, Isil Dillig, and Yu Feng. SolType: Refinement types for arithmetic overflow in Solidity. In *Proc. ACM POPL*, 2022.

[49] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. SmartCheck: Static analysis of Ethereum smart contracts. In *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*, pages 9–16, 2018.

[50] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proc. ACM CCS*, 2018.

[51] Xiao Yi, Daoyuan Wu, Lingxiao Jiang, Yuzhou Fang, Kehuan Zhang, and Wei Zhang. An empirical study of blockchain system vulnerabilities: Modules, types, and patterns. In *Proc. ACM FSE*, 2022.

[52] Xiao Liang Yu, Omar Al-Bataineh, David Lo, and Abhik Roychoudhury. Smart contract repair. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(4):1–32, 2020.

[53] Xiao Liang Yu, Omar Al-Bataineh, David Lo, and Abhik Roychoudhury. Smart contract repair. *ACM Trans. Softw. Eng. Methodol.*, 29(4), sep 2020.

[54] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais. SoK: Decentralized finance (DeFi) incidents. In *Proc. IEEE Symposium on Security and Privacy*, 2023.

Appendix

A False Positive Analysis

Among the 31 false positives mentioned in §6.2, we have classified them into two categories based on their root causes:

Check without Revert (13 of 31). False positives in this category arise from the check extraction process. During our analysis, we extract the checks from the `require`, `if-revert`, or `modifier` statements. For the `if-revert` checks, if the function does not revert the exception, ZepScope does not identify it as a check. For example, the `safeMint` function in the contract `LifeForms` [12] uses the `if` statement to judge the existence of the `tokenId`, which can not be considered as a check in ZepScope. This leads to a false positive.

Check in Equivalent Format (18 of 31). This category encompasses false positives resulting from developers implementing equivalent checks. While we've taken into account numerous equivalent role checks in §5, corner cases remain. An illustrative example is the `lzReceive` function in the `Kev0` contract [11]. It checks roles by evaluating if `msg.sender` matches a specific, authenticated address. Since we cannot conclusively ascertain that the judgment tied to `msg.sender` is authentic, we cannot confirm that the role check is secure. This results in a reported false positive.

B Figures of Two Case Studies

```
function transferOwnership(address newOwner) public virtual {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    emit OwnershipTransferred(owner, newOwner);
    owner = newOwner;
}
```

Figure 11: Case study 1 from the contract A.

```
function burn(uint _id) public {
    _burn(_id);
}
function _burn(uint256 tokenId) internal virtual {
    address owner = ERC721.ownerOf(tokenId);
    _beforeTokenTransfer(owner, address(0), tokenId);
    owner = ERC721.ownerOf(tokenId);
    delete _tokenApprovals[tokenId];
    unchecked {_balances[owner] -= 1;}
    delete _owners[tokenId];
    emit Transfer(owner, address(0), tokenId);
    _afterTokenTransfer(owner, address(0), tokenId);
}
```

Figure 12: Case study 2 from the contract B.

Table 4: The impact of each step of similarity matching.

Component	Ethereum	BSC	Polygon
ZepScope (baseline)	16,873	14,444	16,114
W/O Message Matching	17,183	14,538	16,345
W/O Check Matching	64,278	57,956	66,178

Table 3: The impact of the threshold T_{error}.

T_{error}	Ethereum	BSC	Polygon	Average
0.8	16,842	14,410	16,108	15,787
0.9	16,873	14,444	16,114	15,810

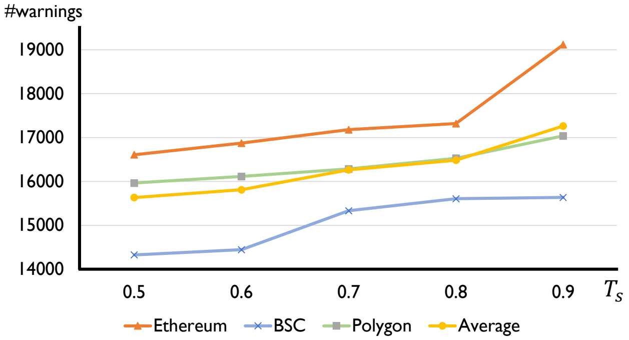


Figure 13: The impact of the threshold T_s.

C Parameter Analysis

C.1 Impacts of the Threshold

In §5.3.1, we introduced two thresholds: T_{error} and T_s. Here we further investigate how different threshold values could impact the final results. For T_{error}, as shown in Table 3, we evaluated two major threshold values, 0.8 and 0.9. We can see that both values of T_{error} present similar warning numbers. Since we need to be conservative about T_{error}, we eventually chose $T_{error} = 0.9$ for our experiments. For the threshold T_s, we explored values ranging from 0.5 to 0.9, with the outcomes illustrated in Figure 13. Notably, the number of warnings increases sharply when T_s moves from 0.6 to 0.7. To avoid excessive false positives, we set T_s to 0.6 in our experiments.

C.2 Impacts of the Dynamic Factor

As described in §5.3.2, we introduced a dynamic factor F to enforce structured similarity matching across the varying lengths of checks. Here we compare the performance of this dynamic factor against different fixed factors, with results presented in Figure 14. We can see that the dynamic factor consistently yielded more warnings than the fixed factors, demonstrating its ability to adapt to varying lengths of checks. For example, setting a fixed factor at 0.5 requires satisfying only 2 out of 4 elements, which might overlook critical errors. Conversely, a fixed factor of 0.9 demands that all elements be met, even when there are as many as 5 checks, potentially resulting in an increase in false positives. In contrast, the adaptability of the dynamic factor allows for better performance, effectively balancing different scenarios.

D Ablation Study on Similarity Matching

When detecting insecure OpenZeppelin code in §5.3, we introduced a two-step similarity matching process to compare the code checks with the facts mined from the OpenZeppelin library. Here, we conduct an ablation study to evaluate the impact of each step in the similarity matching process. As shown in Table 4, since the error message matching step serves as a pre-filter to reduce mismatches in the following step and has a high threshold, removing it slightly increased the number of warnings. In contrast, the check matching step is the core of the similarity matching process, responsible for the majority of the warnings. Hence, removing it significantly increased the number of warnings. The two steps are complementary to each other and both are essential for the final results.

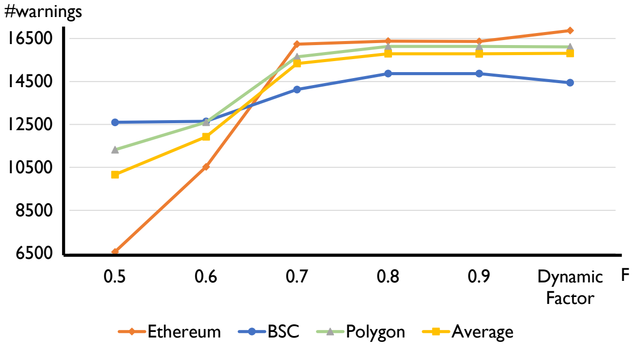


Figure 14: The dynamic factor F vs. different fixed factors.