Demystifying OpenZeppelin's Own Vulnerabilities and Analyzing Their Propagation in Smart Contracts

Han Liu*, Daoyuan Wu†§, Yuqiang Sun‡, Shuai Wang*, Yang Liu‡, Yixiang Chen¶
*The Hong Kong University of Science and Technology, Hong Kong SAR, China
†Lingnan University, Hong Kong SAR, China
‡Nanyang Technological University, Singapore
¶East China Normal University, Shanghai, China
Emails: liuhan@ust.hk, daoyuanwu@ln.edu.hk, suny0056@e.ntu.edu.sg, shuaiw@cse.ust.hk,
yangliu@ntu.edu.sg, chenyx61@aliyun.com

***Abstract*—OpenZeppelin is a building block for many smart contracts on Ethereum-compatible blockchains. It provides modular and reusable libraries for various Ethereum standards (e.g., ERC20 and ERC721) and common functionalities such as upgradeable contracts. Little research has been done on OpenZeppelin security except for a recent study, which focused only on the *misuse* of OpenZeppelin code, assuming OpenZeppelin itself is secure but contract developers may not follow OpenZeppelin's function checks appropriately. We argue that, despite appearing robust, OpenZeppelin itself could have many vulnerabilities, and these library-level vulnerabilities could inadvertently affect third-party smart contracts, even without misuse from developers.**

We present ZEPCOMPARE, the first end-to-end system for demystifying OpenZeppelin's own vulnerabilities and analyzing their propagation in third-party smart contracts. ZEPCOMPARE incorporates a manual analysis stage where we review OpenZeppelin's 64 historical releases, identifying 109 vulnerable-fixed code pairs, exposing flaws in cryptographic utilities, access control, etc. Leveraging these pairs, ZEPCOMPARE introduces *facts of changes*, a novel structure capturing vulnerable and fixed code contexts for flexible matching. Evaluated across 88,605 contracts from three Ethereum-compatible chains, ZEPCOMPARE detects 4,708 instances of OpenZeppelin-derived vulnerabilities. Manual sampling and a ground-truth experiment confirm that ZEPCOMPARE achieves 86.7% precision and 77.1% recall. Our findings reveal significant security risks in both historical and the latest versions of OpenZeppelin libraries, underscoring the urgent need for systematic auditing of foundational contracts components.

***Index Terms*—OpenZeppelin Library, Smart Contracts, Vulnerability Propagation, Vulnerability Detection.**

I. INTRODUCTION

Smart contracts are self-executing digital agreements encoded on a blockchain [1], automatically enforcing terms upon predefined conditions and eliminating intermediaries. Leveraging blockchain's inherent transparency and security, they facilitate trustless execution without central authorities, leading to widespread adoption in finance [2], [3]. However, they remain susceptible to vulnerabilities, resulting in significant financial losses; security attacks had caused cumulative losses of $9.04 billion USD by November 2024, posing a major challenge to the ecosystem and its users [4].

To standardize development and reduce vulnerabilities in smart contracts, OpenZeppelin [5] provides a set of widely used libraries that include common math functions [6], upgrade mechanisms [7], access control components [8], and implementations of ERC standards such as ERC20 [9]. Despite the vital importance of OpenZeppelin to Ethereum and its EVM-compatible chains [10], little academic research has focused on the security of OpenZeppelin. The only work we are aware of is ZepScope [11], which analyzed the security checks in the official OpenZeppelin libraries and whether they are faithfully enforced in the relevant OpenZeppelin functions of real contracts. As a result, ZepScope detected only the *misuse* of OpenZeppelin code, assuming that OpenZeppelin itself is secure but contract developers may not follow OpenZeppelin's function checks appropriately. While valuable, this work assumes OpenZeppelin's inherent correctness, focusing solely on developer errors rather than flaws in the library itself.

We challenge this assumption and argue that OpenZeppelin —despite its widespread trust and adherence to security best practices like checks-effects-interactions [12]—may introduce vulnerabilities into dependent contracts. Indeed, OpenZeppelin has undergone multiple audits and reports only 19 CVEs since 2015 [13]. More importantly, its role as a shared dependency amplifies risks: even minor flaws could propagate to thousands of contracts across EVM chains. Our work tries to address this blind spot, examining how vulnerabilities in OpenZeppelin's codebase, rather than mere misuse by developers, could systematically undermine blockchain ecosystems. This distinction is critical, as library-level flaws require fundamentally different mitigations than developer errors, necessitating a paradigm shift in smart contract security practices.

However, it is a challenging task to explore the OpenZeppelin's own vulnerabilities and their propagation in third-party contracts. (*i*) OpenZeppelin's vulnerabilities span diverse types and code patterns—from cryptographic flaws in mathematical utilities to access control bypasses—with no unified taxonomy to guide detection. This heterogeneity renders rule-based detectors ineffective, as they rely on predefined heuristics ill-suited for novel or undocumented vulnerabilities. (*ii*) Unlike other languages, developers in the Ethereum ecosystem often customize OpenZeppelin libraries to suit their needs (we will discuss a motivating example in Section II-B), leading to a wide range of variations in the code. Additionally, OpenZep-

[§] Corresponding author: Daoyuan Wu.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00083

pelin itself has also provided many customization options, e.g., the `_authorizeUpgrade` function [14] in contract. (*iii*) Although the code is not customization, employing clone-based method [15], [16] to detect it is difficult. There are no existing well-documented datasets of OpenZeppelin libraries and their vulnerabilities. OpenZeppelin library only contains 19 CVEs and does not cover any issues in the earlier versions.

To address these challenges, we propose ZEPCOMPARE, an end-to-end system designed to demystify OpenZeppelin's own vulnerabilities and their propagation in third-party smart contracts. ZEPCOMPARE operates in two phases: in the offline phase, it extracts OpenZeppelin's library-level vulnerabilities as a one-time effort, and in the online phase, it detects the extracted vulnerabilities in given smart contracts. The offline phase begins with a manual review of OpenZeppelin's release history to identify security-relevant code changes and their entry points—functions through which vulnerabilities manifest. Then, the core problem lies in translating these code changes into actionable detection rules. Traditional approaches [15], [16], which rely on syntactic or semantic code comparisons, fail here due to the pervasive customization and semantic complexity of vulnerabilities.

To overcome this, we introduce a novel structure called *facts of changes* (formally defined in Section II-C), which covers not only the affected functions and their contexts but also the structured content of both vulnerable and fixed code supporting for flexible matching. Based on this concept, during the offline phase, (*i*) extracts facts of changes that encapsulate both entry functions and specific code alterations, and (*ii*) extends these facts via an alias analysis for identifying customizations in real-world contracts. During the online phase, ZEPCOMPARE (*i*) identifies the target functions in smart contracts, (*ii*) extracts all of their statements, including function calls as inlined statements, and (*iii*) conducts structured matching between the extracted contract statements and OpenZeppelin's facts of changes by matching vulnerable versions and excluding fixed versions. It flexibly verifies the presence of key vulnerable code patterns while simultaneously checking for mitigations introduced in patched OpenZeppelin versions.

With ZEPCOMPARE's offline module, we identify 109 pairs of OpenZeppelin vulnerable code with functions among the 64 versions of OpenZeppelin libraries. Through manual correlation with the Smart Contract Weakness Classification (SWC) [17], we categorize these vulnerabilities into 20 categories, with access control-related issues being the most prevalent, accounting for 34 of the 109 detected vulnerabilities. We further analyze the characteristics of these vulnerabilities. Our analysis offers a systematic categorization of well-documented vulnerabilities and examines their defining characteristics. (See Section VI-A for more details).

To evaluate ZEPCOMPARE's detection capability, we conduct both a ground-truth experiment comprising 35 real-world OpenZeppelin-related bugs and a large-scale cross-chain experiment targeting 88,605 smart contracts across three Ethereum-compatible chains, including BSC, Ethereum, and Avalanche. Our ground-truth experiment, compared with ZepScope [11], Slither [18], SmartCheck [19], Mythril [20], Manticore [21], and Sun *et al.*'s method [16], reveals that ZEPCOMPARE significantly outperforms these state-of-the-art (SOTA) tools and methods. Specifically, ZEPCOMPARE successfully detects 27 out of 35 ground-truth vulnerabilities, achieving a recall of 77.1%. In comparison, Sun *et al.*'s method and SmartCheck detect 14 and 8 vulnerabilities, while Slither and ZepScope each identify only 5, and neither Mythril nor Manticore identifies any issues. We also analyze the causes of the failed cases for all these tools in Section VI-B. On the other hand, our large-scale experiment flags 4,708 contracts as potentially containing vulnerabilities originating from OpenZeppelin libraries, meaning that, on average, only 5.3% of the contracts require further manual verification. By randomly sampling 100 flagged contracts from each blockchain, we confirm 260 as true positives and 63 as false positives, achieving an average precision of 86.7% across the three blockchains. From the large-scale experiment, we analyze the prevalence and scope of the issues and identify 16 vulnerable contracts with a total balance of $765,031 by focusing on several high-value contracts. The artifacts of this paper are available at [22].

In summary, we make the following contributions:

- (*Systematization*) We conduct the first systematic study of vulnerabilities in OpenZeppelin library across all versions and analyze their propagation in real-world contracts.
- (*Methodology*) We propose ZEPCOMPARE, a novel tool for mining security-related changes in OpenZeppelin library and detecting their propagation in real-world smart contracts. ZEPCOMPARE employs a novel structure called *facts of changes* to extract the security-related changes in OpenZeppelin library and detect their propagation in real-world smart contracts.
- (*Evaluation*) Experiment on the ground-truth dataset shows that ZEPCOMPARE outperforms SOTA tools in detecting OpenZeppelin-related bugs. Large-scale experiments on 88,605 smart contracts from three blockchains show that ZEPCOMPARE has a high accuracy with 86.7% and found 16 bugs in the smart contracts.
- (*Insights*) We provide an understanding of the vulnerabilities in OpenZeppelin library, a comprehensive list of 109 pairs of vulnerable code with details, and a qualification of the propagation of these vulnerabilities in real-world smart contracts.

II. PRELIMINARY AND DEFINITIONS

A. OpenZeppelin Libraries

OpenZeppelin [5] is a widely used open-source framework for developing secure smart contracts on the blockchains [1]. It provides modular and reusable libraries that implement key Ethereum standards, such as ERC20 [9] for fungible tokens and ERC721 [23] for non-fungible tokens. By following industry best practices and undergoing security audits, OpenZeppelin helps mitigate common vulnerabilities, supporting the development of secure decentralized applications (DApps) [3].

Nevertheless, OpenZeppelin is not perfect. Over the years, OpenZeppelin has evolved significantly, with 64 versions

```
1  function recover(bytes32 hash, bytes memory
       signature) internal pure returns (address) {
2      ...
3      if (signature.length != 65) {
4          return (address(0));
5      }
6      assembly {...
7      v:=byte(0,mload(add(signature , 0x60))}
8      // Inadequate validation of v
9      if (v < 27) {v += 27;}
10     if (v != 27 && v != 28) {
11         return (address(0));
12     } else {
13         return ecrecover(hash, v, r, s);
14 }
```

Fig. 1. The vulnerable `recover` function in OpenZeppelin ECDSA library (version: <= 2.1.3).

released to date. Despite 19 CVEs and GitHub security advisories [13], our research reveals a more extensive security landscape. For example, while the oldest officially recorded CVE is in version 4.3.0 of the OpenZeppelin library [24], the example in Section II-B shows an attack incident caused by a vulnerability in OpenZeppelin as early as version 2.1.3, indicating that security risks predate common acknowledgment.

B. A Motivating Example

We use an example to illustrate how OpenZeppelin could be vulnerable in one of its core cryptography libraries and how this vulnerability could lead to a real-world attack, as reported by DeFi Hacks [25]. As shown in Fig. 1, OpenZeppelin's ECDSA library introduced a vulnerable `recover` function in version 2.1.3 and earlier, which contains a flaw in the signature validation logic. Specifically, on line 9, the code allows the `v` value, derived from the incoming `signature` on line 7, to be adjusted by adding 27 when `v` is less than 27. This adjustment permits signatures with `v = 0` or `v = 1` to be transformed into valid signatures with `v = 27` or `v = 28`. This introduces a vulnerability called Signature Malleability [26], allowing an attacker to create multiple valid signatures for the same message, which can lead to exploits.

This vulnerability of OpenZeppelin, unfortunately, caused an attack incident in a contract named `TCHtoken` in May 2024 [27]. As shown in Fig. 2, although developers made certain customizations when reusing OpenZeppelin libraries, the core vulnerable logic still propagated from OpenZeppelin to `TCHtoken`. Specifically, `TCHtoken` also allowed two types of `v`, including 0,1 and 27,28. The attacker harvested previously submitted signatures and modified the `v` part of the signature: instead of submitting 0x01 (1), they submitted 0x1c (28). Since at this point `v` is already greater than 27 (line 9), there is no need to add 27 to `v`, resulting in the same outcome as when `v = 0x01`. As a result, the fake signature was successfully verified with `ecrecover` on line 2. The attacker ultimately burned a large number of TCH tokens owned by the PancakeSwap [28] pair, which allowed him/her to manipulate the price in the pool and take the profit.

From this example, we observe that despite significant efforts to secure OpenZeppelin—for example, a $1M USD grant was awarded in January 2023 [29]—OpenZeppelin libraries can still contain serious vulnerabilities, which affect real-world smart contracts, whether in the original OpenZeppelin format or in customized formats. In addition, the 2.1.3 version of the ECDSA library was released in February 2019, and the attack incident happened in June 2024, indicating that the vulnerability persisted for a long time before being exploited. This motivates us to develop a system to demystify OpenZeppelin's own vulnerabilities across different versions and the propagation of these vulnerabilities in third-party contracts.

```
1  function recoverSigner(bytes32 ethSignedMessage,
       bytes memory signature) internal pure
       returns (address) {
2      (bytes32 r, bytes32 s, uint8 v) =
           splitSignature(signature);
3      return ecrecover(ethSignedMessage,v,r,s); }
4  function splitSignature(bytes memory sig)
       internal pure returns (bytes32 r, bytes32 s,
       uint8 v) {
5      require(sig.length == 65, "Invalid signature
           length");
6      assembly {...
7      v:=byte(0, mload(add(sig, 96)}
8      // Inadequate validation of v
9      if (v < 27) v += 27;
10     return (r, s, v);
11 }
```

Fig. 2. The vulnerable `recoverSigner` function in `TCHtoken`.

C. The Definition of Facts of Changes

We define a *Fact of Change* as a structured entity represented by the triple $FC = (S_{ent}, C_{vul}, C_{fix})$ to capture the essence of a change made to a function in the OpenZeppelin library. The components of this definition are as follows: S_{ent} denotes the *Entry Function Signature*, which comprises the contract name C_{name}, function name F_{name}, parameter types P_{types}, return value types R_{types}, emitted events E, read/write constants RW, and a boolean indicator P specifying whether the function is public.

The components C_{vul} and C_{fix} are fact contents, representing the *Two Changed Sequences* for the vulnerable and fixed versions of the function, respectively. Each fact content is an ordered list of *Fact Units*, denoted as $C = [F_1, F_2, \ldots, F_n]$, where the ordering reflects the sequence of statement definitions within the function. Each Fact Unit F_i corresponds to a specific statement in the code and is categorized into one of seventeen distinct statement types. For general statements, F_i is modeled as a heterogeneous binary tree BTree(O, L, R), where O is the operator at the root, and L and R are the left and right subtrees representing subsequent operators. In cases involving function calls (FC), variable assignments (VA), or equivalent variables (EV), F_i is represented as an array $[F_{i1}, F_{i2}, \ldots, F_{ik}]$, containing multiple equivalent facts to account for variations in code constructs.

Formally, the Fact of Change is expressed as:

$$FC = (S_{ent}, C_{vul}, C_{fix})$$

where

$$S_{ent} = (C_{name}, F_{name}, P_{types}, R_{types}, E, RW, P)$$

and

$$(C_{vul}, C_{fix}) = \big([F_{vul1}, F_{vul2}, \ldots, F_{vuln}],\\ [F_{fix1}, F_{fix2}, \ldots, F_{fixm}]\big) \tag{1}$$

Each Fact Unit F_i is defined as:

$$F_i = \begin{cases} \text{BTree}(O, L, R) & \text{if the statement} \notin \{\text{FC, VA, EV}\}, \\ [F_{i1}, \ldots, F_{ik}] & \text{if the statement} \in \{\text{FC, VA, EV}\}. \end{cases} \tag{2}$$

III. Overview

Motivated by Section II-B, we aim to achieve the following two main objectives in this paper:

Objective 1: To demystify the security issues in different versions of OpenZeppelin's official libraries and identify the characteristics of these vulnerabilities.

Objective 2: To analyze the propagation of OpenZeppelin's vulnerabilities in third-party smart contracts, including the effectiveness of the detection onOpenZeppelin's security issues and the extent of their impact.

To achieve this, we propose ZepCompare, an end-to-end system for demystifying OpenZeppelin's own vulnerabilities and analyzing their propagation in smart contracts. Fig. 3 presents an overview of ZepCompare, consisting of the offline phase (detailed in Section IV) and the online phase (detailed in Section V). Offline phase conducts a one-time effort to demystify the vulnerabilities from the OpenZeppelin library, and converts them into a structured format called *facts of changes* (see Section II-C), which are then stored in a database. The online phase, on the other hand, is a multi-time effort for detection of the OpenZeppelin's own vulnerabilities in third-party contracts. It takes the facts of changes as input and conducts a signature pre-filtering step to identify the functions that have similar functionalities. Then, it extracts the code facts from the function and performs a structured matching to determine whether the vulnerable code is present in third-party contracts.

IV. Demystifying OpenZeppelin's Vulnerabilities

In this section, we introduce ZepCompare's offline module, which demystifies OpenZeppelin's own vulnerabilities by extracting them as *facts of changes* (see Section II-C) in a one-time effort. As shown in Fig. 3, ZepCompare completes three tasks during this offline phase: identifying security-related code changes, extracting and extending facts of changes. We present these tasks in the following subsections.

A. Identifying Security-Related Code Changes

In order to analyze OpenZeppelin's vulnerabilities, we need to identify the security-related code changes in the OpenZeppelin repository across different versions. Although OpenZeppelin employs GitHub to manage its code, the commit-level diffs do not directly reflect the original vulnerable and patched code in the releases. Hence, we can not directly use existing commit-level security changes identification methods [30]–[33] to identify the security-related code changes in OpenZeppelin. Additionally, it is important to have a higher confidence in the correctness of the mined vulnerabilities, as they will be used to analyze the propagation of vulnerabilities in real-world smart contracts. Hence, as illustrated in Fig. 3, we first conduct a manual review of the release notes with the pull request (PR) descriptions and security advisories to identify all the security-related code changes in OpenZeppelin. Three authors independently review OpenZeppelin's release notes, PR descriptions, and security advisories to identify candidate security-related code diffs. Then, the authors compare their findings. In cases where there is disagreement about whether a code diff is security-relevant, the authors discuss the specific case together. If consensus can not be reached through discussion, the majority opinion is adopted.

To further map the commit-level diffs to the original releases, we construct an order of the versions based on the release time and the versioning scheme. We then map the commit-level diffs to the related releases, ensuring that the identified code changes are indeed officially released. For each identified code change, we find all the entry functions, i.e., S_{ent} as defined in Section II-C, that are relevant to the code changes for subsequent analysis. Finally, we totally identify 109 pairs of entry functions and their corresponding code changes from 938 candidates in the OpenZeppelin repository.

B. Extracting Facts of Changes

From the located entry functions and code change statements, we can now extract their signatures and facts based on the definition given in Section II-C.

Extracting Function Signature. For each identified target statement, we extract its function characteristics for subsequent recognition. In our definition, we use the Entry Function Signature S_{ent} to represent these function characteristics, comprising a total of eight elements. First, we record the contract name, function name, input parameters, return parameter types, and whether the function is public or external. Next, we traverse the function's statements and record any events or accesses to state variables and Solidity variables. This process ultimately forms the signature of the entry function.

Extracting Facts Content. To extract the facts from the statement, we divide the facts into several types, i.e., function entry point statement, expression statement, loop statement, require and return statement, if-revert statement, common condition statement, function call statement, new variable statement, and other statement. These selected fact types are chosen to comprehensively capture the semantic and structural elements of smart contract logic. We now further present each as follows.

- *Function Entry Point Statement*: The function entry point statement is recorded as a fact unit. We record the entry function's visibility in the left node of the fact unit. Visibility is grouped into two categories: external/public and private/internal. The right node records the name of the

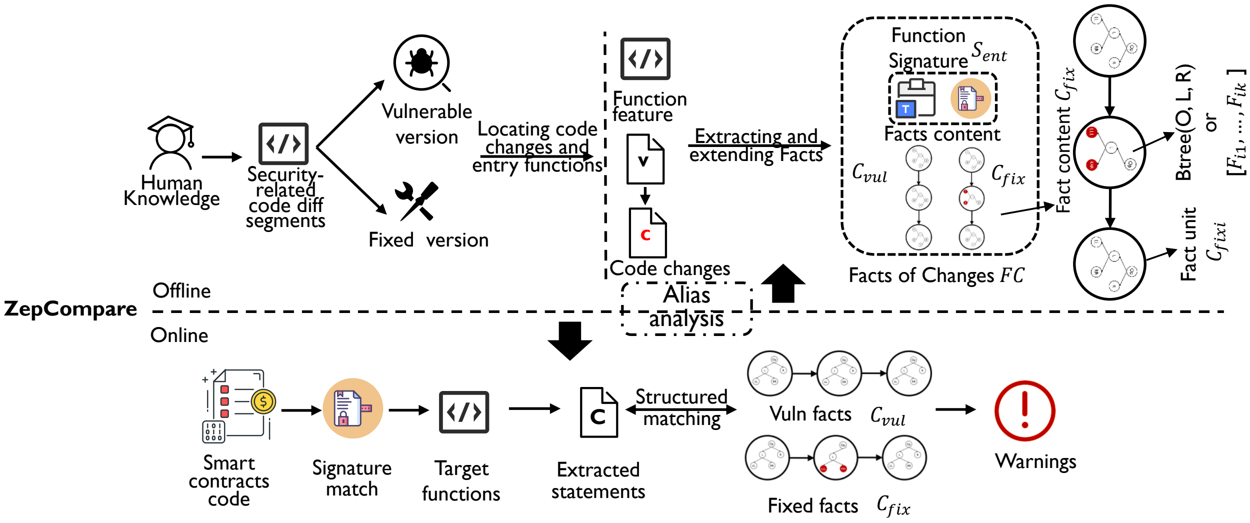

Fig. 3. An end-to-end system for demystifying OpenZeppelin's vulnerabilities and analyzing their propagation in smart contracts.

modifier, extended with additional elements extracted from the modifier's content.

- *Expression Statement*: The expression statement is also recorded as a fact unit. We recursively process the expression as follows: (*i*) For binary operations, index access, tuple, member access, or assignments, divide into left and right expressions, recording the operation as the root. (*ii*) For unary operators or type conversions, process sub-expressions as the left node, recording the operation/type conversion as the root. (*iii*) For variables or literals, capture the variable name as the left node, and record `variable` as the root. (*iv*) For other expressions, capture the stringified statement, recording `other expression` as the root.
- *Loop Statement*: The loop statement is recorded as an ordered sequence of fact units. We first use the loop and end-loop fact units to define their boundaries. Then, we record the loop condition as the first fact unit, followed by the loop body. For the loop body, we recursively process the statements using the extracting method described in this section. Finally, we record the loop statement as an ordered sequence of fact units.
- *Require and Return Statements*: The require and return statements are recorded as fact units. We capture and recursively process the expression as the left node and record `require` or `return` as the root. For require statements, we also capture the error message in the right node.
- *If-Revert Statement*: The if-revert statement is recorded as a fact unit. We identify the corresponding if expression, process it as the left node, and carefully analyze the control flow. For `revert` in if or else-if conditions, we invert the logic. For `else` conditions, we retain the original logic. Finally, we record the revert statement as the right node and `if-revert` as the root.
- *Common Condition Statement*: The common condition statement is recorded as two equivalent ordered sequences, e.g., for if condition A else B, the representation becomes:[[if condition do A; if not condition do B], [if not condition do B; if condition do A]]. For the sub-statements, we recursively process them using the extracting method described in this section.
- *Function Call Statement*: The function call statement is recorded as a fact unit. We capture the called function name as the left node and set the root as `function call`. Then, we process each function argument using the expression statement method, adding the list to the right node.
- *New Variable Statement*: The new variable statement is recorded as a fact unit. We capture the stringified variable. Considering flexibility (e.g., `msg.sender` used directly), we further extend it by alias in Section IV-C.
- *Other Statement*: Other statements are recorded as fact units. We directly capture the stringified statement in the left node, and record `Other` as the root.

C. Extending Facts via Alias Analysis

As described in Section IV-B, we further extend the facts using variable aliases. Not only do facts with new variable statements require extension, but other statements involving variables also need to extend the facts. Thus, ZepCompare performs inter-procedural alias analysis during facts extraction. Specifically, ZepCompare first inlines all function calls in the OpenZeppelin code. It then traverses each function's statements, processing assignment statements and capturing the variables on both sides as pairs of equivalent variables. For right-hand values, we process the expression statement using the method described in Section IV-B. For left-hand values, we only record the stringified value. Additionally, we extend alias analysis from the statement level to the procedural level, capturing arguments and parameters as pairs of equivalent variables. Alias analysis also propagates between different sets that share the same variable. For example, if A and B are equivalent, and B and C are equivalent, then A and C are also equivalent. By recursively extending the equivalent variables, we obtain the complete set of equivalent variables.

Based on the set of equivalent variables, we extend the facts. We perform a pre-order traversal to visit each node of the BTree in the fact unit. If a node has an equivalent variable in our set, we replace the node with a set containing both the node and its equivalent variables. This process is repeated until no further changes occur, at which point we obtain the extended fact.

V. Detecting Their Propagation in Contracts

In this section, we introduce ZepCompare's online module, which detects the propagation of OpenZeppelin vulnerabilities in real-world smart contracts by correlating contract code with *facts of changes*. As shown in Fig. 3, ZepCompare completes three tasks during this online phase: identifying target functions in smart contracts, extracting statements from target functions, and performing structured matching against both vulnerable facts C_{vul} and C_{fix} using the extracted statements. We present them in the following subsections.

A. Identifying Target Functions

The first step in online detection is identifying target functions, which involves locating code segments that perform the same functionality as specific facts. Note that this step serves only as a preliminary screening; the subsequent matching step will further ensure the overall effectiveness of the detector. Our goal is to identify as many functions with potentially similar functionality as possible. We start by analyzing function names. Special characters, such as underscores ("_"), that appear at the beginning of function names are removed. We then account for two common naming conventions: camel case and snake case. We tokenize the function names based on these conventions, generating a set of words for each function name. Subsequently, we employ Word2Vec [34] to convert these words into vector representations and calculate their cosine similarity. If the similarity between the target function name and the function name recorded in the extracted fact exceeds a predefined threshold, we proceed to the further matching step. In this paper, the threshold is set at 0.7, which is chosen empirically by testing it on 100 pairs of OpenZeppelin functions and their variants. Setting a higher threshold resulted in some semantically similar functions being filtered out, while a lower threshold introduced more unrelated functions. Therefore, we chose the threshold that best balanced these trade-offs.

We further refine the preliminary results by examining additional function signatures. We require that the parameters of the target function include at least those in the fact, ensuring that all parameter types in the fact are present in the target function. If developers have altered parameter type names (e.g., from ERC20 to ERC20Burnable), we consider the parameters to match if there is a substring relationship between the two types (excluding single-letter parameter types). We apply the same method to check the return types, emitted events, and read/write access to constants. If all of these signatures match, we designate the function as a target function and proceed with further extraction and matching.

B. Extracting Statements of Target Functions

After identifying the target functions, we extract all statements within the target functions for further matching. Unlike the extraction process in the offline phase, where we precisely identified target statements, our approach here involves locating entire target functions that may contain numerous statements. Therefore, we extract the entire function content for subsequent matching. Since the statements in the facts are ordered, we first sort the statements within the function according to their code sequence.

Unlike in Section IV-B, in this phase, we do not perform equivalence processing on function call statements. Instead, when encountering a function call, we inline the called function's content after the call. This ensures that in the subsequent analysis, whether we match the vulnerable function call or specific content within the function call, we can consider the function to contain a vulnerability in OpenZeppelin. For the remaining parts, we follow the procedure outlined in Section IV-B to extract statements for the eight types of statements and obtain the corresponding function content for structured matching. Additionally, in this step, we extend the statements with inter-procedural alias analysis to enhance the robustness of our detection across various smart contracts in the wild.

C. Structured Matching

We proceed to compare the extracted statements from the target function with the specific facts. As shown in Fig. 3, the matching process involves two main steps: matching the vulnerable facts C_{vul} and excluding the fixed facts C_{fix}.

Matching Vulnerable Facts. First, we match the vulnerable facts with the extracted statements. Given that both our facts and extracted statements are sequential structures, we iterate through the fact units within a specific fact and search for each fact unit in the extracted statements. When a fact unit is found, we continue searching for the next fact unit, starting from the position in the extracted statements. This process continues until either all fact units are successfully matched or a mismatch occurs for any fact unit. This approach allows developers with flexibility in modifying the code while ensuring that the vulnerable code remains in a certain order.

However, since our fact units form an irregular tree structure, simple equality checks are not sufficient. Therefore, we employ a novel structured matching method. We recursively evaluate matches according to the following rules: (*i*) If the fact unit represents an equivalence structure, we check whether any of its sub-fact units match the extracted statement. (*ii*) If the fact unit has a binary tree structure, we first check if its root node matches the extracted statement unit. If it does not match, we proceed to match the left and right subtrees. If it matches, we then separately match the left and right nodes of both the fact unit and the extracted statement. (*iii*) If the fact unit and the extracted statement involve `require` and `revert` statements, we convert the `require` statement into a `revert` statement based on logical negation principles and perform a secondary match. (*iv*) If the fact unit involves a binary comparison and the comparison initially fails, we proceed with a secondary comparison by reversing the logic (e.g., after a failed $a > b$ comparison, we also verify $b <= a$).

In structured matching, we ultimately need to determine whether two strings are related. These strings may represent function names or variable names and contain semantic information. However, since some variable names could be simple abbreviations, matching them can still be challenging. To address this, we devised a novel string-matching method that combines tokenization with a lightweight semantic similarity method which are more suitable for our task. Specifically, we first tokenize the strings according to two naming conventions: camel case and snake case. Since the resulting tokens tend to be quite short, existing pre-trained models may fail to capture their semantics, while more complex models could be too time-consuming, especially given the large number of matching processes required during our detection. Therefore, we first compute the similarity between each pair of tokens using edit distance. Then, we apply an improved Longest Common Subsequence (LCS) algorithm [35] to check whether one set of tokens is a subset of the other. If so, we consider the two strings a match; otherwise, they are not matching.

With the above three strategies, we can determine whether an extracted statement matches the fact corresponding to the same target function.

Excluding Fixed Facts. Up to this point, we have preliminarily determined whether the target function contains vulnerable code from OpenZeppelin. However, some vulnerability fixes involve adding additional checks, and our facts only capture certain function signatures for identification. Additionally, the fixed code may be quite similar to the vulnerable code. Therefore, we incorporate matching of fixed code as a supplement to reduce false positives. Specifically, when ZEPCOMPARE identifies vulnerable code, it further uses the extracted fixed facts to match the statements. As described above, we repeat the process to determine whether the fixed fact matches the statements. If a match is found, we consider it to be fixed code and do not issue a warning. Otherwise, we report the presence of vulnerable code.

VI. EVALUATION

This section addresses the two objectives outlined in Section III through our research questions (RQs):

Objective 1: (*Analyzing OpenZeppelin's Vulnerabilities*):

- **RQ1:** What security issues are present in different versions of OpenZeppelin's official libraries, and what are the characteristics of these vulnerabilities?

Objective 2: (*Detection and Real-World Impact*):

- **RQ2**: How effective is ZEPCOMPARE in detecting OpenZeppelin's vulnerabilities in a set of ground-truth vulnerable smart contracts compared to other SOTA static analysis tools?
- **RQ3**: How accurate and efficient is ZEPCOMPARE in analyzing a large set of on-chain smart contracts?
- **RQ4**: To what extent are vulnerabilities stemming from OpenZeppelin libraries prevalent and impactful across heterogeneous on-chain smart contracts?

TABLE I
CATEGORIES OF OPENZEPPELIN'S VULNERABILITIES.

Categories	Count
NO-SWC: Missing Authorization Validation	25
SWC-128: DoS with Block Gas Limit	10
SWC-104: Unchecked Call Return Value	10
SWC-132: Unexpected Ether Balance	8
SWC-124: Write to Arbitrary Storage Location	8
SWC-114: Transaction Order Dependence	8
SWC-122: Lack of Proper Signature Verification	6
SWC-101: Integer Overflow and Underflow	6
SWC-107: Reentrancy	5
NO-SWC: Locigal issue	5
SWC-117: Signature Malleability	3
NO-SWC: Uninitialized Contract Vulnerability	2
SWC-135: Code With No Effects	2
SWC-126: Insufficient Data Validation	2
SWC-113: DoS with Failed Call	2
SWC-111: Use of Deprecated Functions	2
NO-SWC: Logical Flaw in Merkle Tree Verification	2
NO-SWC: Storage compatibility	1
NO-SWC: Cross Chain issue	1
SWC-112: Delegatecall to Untrusted Callee	1

Experimental Setting. We implemented ZEPCOMPARE using the robust static analysis framework Slither [18]. The experiments were conducted on a machine with an AMD Ryzen Threadripper 3970X 32-Core Processor and 252 GB of RAM, ensuring the capacity to handle extensive computations.

A. RQ1: Facts Understanding

We present the results obtained from the entire change history of OpenZeppelin libraries in Section IV. Among the total 64 versions of OpenZeppelin libraries tested, we found 109 vulnerabilities and obtained 109 pairs of facts.

Categories. To address RQ1, we categorize the security issues across various versions of the OpenZeppelin libraries. We use the Smart Contract Weakness Classification (SWC) framework [17] as the foundational taxonomy for categorizing vulnerabilities in OpenZeppelin's smart contracts due to its comprehensive and widely recognized structure. Three authors independently categorized the identified vulnerabilities using the SWC taxonomy, applying a structured card-sorting methodology [36]. When necessary, categories were refined or extended to accommodate emerging or nuanced vulnerability types not covered by existing SWC entries. After the independent phase, the categorizations were compared, and discrepancies were resolved through group discussion. In cases where consensus could not be reached, the majority opinion was adopted to ensure resolution and reliable classification. As shown in Table I, the 109 security issues across 64 versions of OpenZeppelin libraries encompass 20 categories. Given the wide range of potential classifications, we focus primarily on the most prevalent and impactful categories.

Notably, access control-related issues accounted for 34 of the 109 detected vulnerabilities, including *SWC-122: Lack of Proper Signature Verification* [37], *SWC-117: Signature*

Malleability [38], and *Missing Authorization Validation*. These were the most common vulnerabilities in the earlier versions of OpenZeppelin. For example, in version 4.7.1 [39], the `recover` function within the `ECDSA` library initially failed to handle signature malleability correctly, making it vulnerable to *Signature Malleability (SWC-117)* attacks. The issue was fixed in version 4.7.2 by enhancing error handling and enforcing stricter signature parameter validations, mitigating the risk of malleable signatures. As contracts expanded in functionality and complexity across different versions, additional types of vulnerabilities began to emerge. For example, in version 4.9.1, a *Logical Flaw in Merkle Tree Verification* allowed the proof of arbitrary leaves for specific trees. In version 5.0.1, the *SWC-124: Write to Arbitrary Storage Location* vulnerability was identified due to improper handling of writes in dirty memory.

Fix Frequency. To further analyze fix frequency, we systematically review the version history of each affected OpenZeppelin contract. For every identified vulnerability, we trace its initial introduction and examine all subsequent versions to determine when and how the issue is addressed. This process allows us to identify recurring vulnerabilities within the same contracts and to analyze the evolution of related code changes over time, offering deeper insights into the lifecycle of security fixes. The results show that most security fixes in a contract can be related to other changes, often aimed at reducing the attack surface. For example, in the case of `Initializable`, a reentrancy issue was identified in version 4.4.0, and by version 4.9.3, OpenZeppelin had strengthened security by considering storage compatibility in subsequent updates to improve robustness.

For some contracts, due to their complexity, issues may persist even after vulnerabilities are fixed. For example, the `recover` function in the `ECDSA` library underwent signature-related fixes in three separate versions: 2.1.3, 2.5.0, and 4.7.1. These fixes ranged from "no longer accepting malleable signatures" to ensuring that the recovered address cannot be the zero address. The final enhancement involved supporting EIP-2098 compact signatures in addition to the traditional 65-byte signature format, enhancing compatibility and flexibility in signature verification. There were also several changes related to the `Governor` contracts. Due to the complexity of `Governor`'s logic, the contract underwent modifications in versions 4.3.3, 4.5.0, 4.7.1, and 4.9.3 to address various issues, including access control, frontrunning, and failed calls.

Fix Complexity. In most cases, security issues in OpenZeppelin are addressed through straightforward code fixes, such as adding missing validation checks. Repeated updates usually occur due to the discovery of new vulnerabilities or logical flaws as smart contracts evolve. However, the current code rarely involves changes to contract architecture or dependencies that would require more complex solutions, potentially affecting the efficiency of these fixes.

B. RQ2: Comparison with the SOTA Tools

In this RQ, we evaluate ZEPCOMPARE against SOTA static analysis tools. This evaluation should address two critical constraints identified in RQ1: (1) ZEPCOMPARE's coverage of 109 vulnerability patterns spanning 20 distinct categories, which necessitates baselines with broad detection scopes, and (2) its support for multi-version analysis of OpenZeppelin libraries, requiring tools that are compatible with diverse Solidity compiler versions.

We select six baseline tools representing three methodological approaches. First, ZepScope [11] serves as a specialized detector for OpenZeppelin misuse patterns, operating under the assumption that the library itself is secure. Second, four general-purpose analyzers—Manticore [21], Slither [18], SmartCheck [19], and Mythril [20]—are selected based on recommendations from reference [40], which highlights their support for several vulnerability types discussed in RQ1. Third, Sun *et al.*'s clone-based method [16] complements these by identifying code replicas of known vulnerabilities. Notably, we exclude legacy tools like those in [15] due to their incompatibility with modern Solidity versions and EVM upgrades. The final baselines include three academic and three industrial tools, chosen for their methodological diversity, detection coverage, and alignment with OpenZeppelin's versioned ecosystem, as recommended in recent literature due to their effectiveness [40].

To validate ZEPCOMPARE's capabilities, we collect 35 real-world security issues related to OpenZeppelin. The datasets are selected based on two main criteria: (1) *Source*: We choose datasets from reputable bug report platforms (i.e., Code4rena [41] and Sherlock [42]) and curated bug collections (i.e., DeFiHackLabs [25], the SmartBugs dataset [43], the Web3Bugs dataset [44]) to ensure the inclusion of real-world, security-relevant cases. (2) *Relevance to OpenZeppelin*: Only contracts related to OpenZeppelin are included. This means the contract either explicitly references OpenZeppelin in its bug analysis or implements logic that is the same as, or very similar to, known vulnerable OpenZeppelin code. All bugs are confirmed by at least two authors to ensure they involved OpenZeppelin-related flaws. These 35 bugs form our ground-truth dataset for in-depth analysis, which we believe is in line with or larger than several prior works [11], [45]–[47].

We analyze the dataset by the tools, recording the True Positives (TP), False Positives (FP), and False Negatives (FN) for each tool. We conduct a function-level root cause comparison: only when both the function and the root cause are correctly matched do we consider a result a TP. Otherwise, it's counted as a FP or a FN. This checking process is performed through manual inspection by two authors independently, with a third author acting as an arbiter in cases of disagreement. Noted that, for Sun *et al.*'s method [16], we provide the vulnerable function in the OpenZeppelin library, and record whether it can detect the vulnerability in the contract.

Table II shows the results of the five tools on our ground-truth dataset. Specifically, ZEPCOMPARE detected 28 out of the 35 security bugs. The remaining 7 were missed mainly due to the following two reasons: (*i*) The detection granularity in ZEPCOMPARE is larger than the issues in the ground-truth dataset. (*ii*) Matching failures. We discuss these reasons in

TABLE II
Results of ZepCompare with six SOTA tools on 35 real-world bugs containing OpenZeppelin's library vulnerabilities.

Tool	TP	FP	FN	#Failed
ZepScope	5	3	29	1
Manticore	0	0	21	14
Sun *et al.*	14	0	21	0
Slither	5	25	30	0
SmartCheck	8	29	27	0
Mythril	0	0	29	6
ZepCompare	27	0	8	0

TABLE III
Evaluation results of ZepCompare on 88,605 contracts across three Ethereum-compatible chains.

Chains	# Contracts	# Success	# Failure	# Warings	Sampled Accuracy
BSC	29,841	29,608	233	337	80%
Ethereum	29,286	29,063	223	2,767	88%
Avalanche	29,478	28,911	567	1,604	92%
Overall	88,605	87,582	1,023	4,708	86.67%

detail on our website [22].

Sun *et al.*'s method [16] performed well, identifying 14 issues. However, it could not support the diverse customization of OpenZeppelin libraries, which is a common practice in real-world contracts. This limitation led to a false negative of 21 issues. Additionally, it needs to point out the specific function of the vulnerability in a huge codebase of OpenZeppelin library, which is also a hard task compared to ZepCompare's automatic detection. The other five tools' detection capabilities significantly lagged behind ZepCompare. ZepScope performed poorly compared to ZepCompare, identifying only 5 issues. This is because ZepScope primarily focuses on extracting and analyzing problems related to `require` and `if-revert` statements. However, many OpenZeppelin vulnerabilities are not solely addressed by these checks. For example, issues like return statement overflows, calculation errors, and signature length problems cannot be detected using `require` or `revert` statements, making ZepScope ineffective for such cases. The other four general static analysis tools also performed poorly. SmartCheck and Slither only identified 8 and 5 security issues with a high false positive, respectively. Mythril and Manticore detect zero of the issues. This was mainly due to (1) These tools lacking rules to detect these specific issues, (2) some OpenZeppelin vulnerabilities requiring consideration of edge cases, and (3) their inability to correctly identify and assess common issues, such as reentrancy and access control, without domain knowledge.

C. RQ3: Accuracy and Performance

In this RQ, we perform an evaluation of ZepCompare's accuracy and performance using a large-scale dataset of on-chain smart contracts. We first chose Ethereum and BSC as the primary chains to collect on-chain contracts because they are the most popular and widely used Ethereum-compatible chains [10], representing the mainstream of the EVM ecosystem. Additionally, we included Avalanche, which is among the top three Ethereum-compatible chains and features a unique architecture [48] to evaluate ZepCompare's effectiveness across different chain designs. Then, we collected the dataset from the top 30,000 contracts of each chain, ranked by balance. Specifically, we retrieved the source code by crawling the official APIs of BSCScan, Etherscan, and Snowtrace. Due to download limits and network issues, we ultimately obtained 29,841, 29,286, and 29,478 contracts from the three chains, respectively. We then compiled and analyzed them. To reduce time expenses, we set 60 parallel processes and a maximum time of 10 minutes for the analysis.

As shown in Table III, ZepCompare successfully analyzed 87,582 contracts and failed on 3,069 contracts. The failures were primarily due to compilation errors and timeouts. Ultimately, ZepCompare flagged 4,708 contracts out of 88,605 as potentially containing vulnerabilities originating from OpenZeppelin libraries. This means that, on average, around 5.3% of the contracts require further manual verification to confirm the presence of insecure code. Given this scale of warnings, manual review is entirely feasible, and ZepCompare generates significantly fewer warnings compared to similar previous work [11].

Accuracy Evaluation. Since no ground truth is available for such a large dataset, and manually verifying 87,582 contracts is unrealistic, we randomly sampled 100 flagged contracts from each chain for detailed manual inspection to assess whether the warnings were false positives. To ensure accuracy, two authors independently reviewed the results, and a third author resolved any disagreements. Out of the 300 sampled contracts, we confirmed 260 as true positives and 40 as false positives. Table III presents the results for each chain.

ZepCompare performed best on the Avalanche chain, with 92% accuracy, while its performance on the BSC chain was slightly lower, with 80% accuracy. The decreased accuracy on BSC is due to more modifications of OpenZeppelin code, such as adding extra protective measures that our tool did not handle well. The primary cause of false positives on other chains was matching errors. To maximize detection, ZepCompare applies relaxed criteria during the signature recognition phase, which sometimes includes unrelated code that still meets the structured matching criteria, leading to incorrect reports.

Overall, ZepCompare's accuracy is acceptable, achieving an average of 86.67% across the three blockchains, which compares favorably to or exceeds similar studies [11], [45], [47]. Moreover, over 94.7% of the contracts raised no warnings, indicating that large-scale deployment and manual review of ZepCompare's warnings are feasible.

Performance Evaluation. Additionally, to assess whether ZepCompare is suitable for large-scale operation, we recorded its runtime on the 87,582 contracts. With 60 parallel processes, ZepCompare took a total of 23,296.81 seconds to analyze all contracts, averaging 15.96 seconds per contract. This demonstrates that ZepCompare is highly efficient, making large-scale contract analysis feasible.

```
1  function R(bytes32 hash, bytes memory sign)
       internal pure returns (address, RecoverError
       ) {
2      if (sign.length == 65) {
3          ...
4          assembly {...}
5          return R(hash, v, r, s);
6      } else if (sign.length == 64) { //
           vulnerable point : Accepting both
           traditional 65-byte signatures and the
           more compact 64-byte EIP-2098 signatures
7          ...
8          assembly {
9              r := mload(add(sign, 0x20))
10             vs := mload(add(sign, 0x40))
11         }
12         return R(hash, r, vs);
13     } else {... // Raise Error}
14 }
```

Fig. 4. The vulnerable `tryRecover` function in case study 1, originating from the `ECDSA` contract of OpenZeppelin libraries (version ≤ 4.7.1).

D. RQ4: Prevalence and Impact

We investigate the pervasiveness and real-world consequences of vulnerabilities originating in OpenZeppelin libraries across heterogeneous blockchain platforms. By analyzing 88,605 on-chain contracts flagged by ZEPCOMPARE (Section VI-C), we quantify ecosystem risks and trace vulnerability propagation as the following three key findings:

(*i*) ***Cryptographic Primitives Are High-Risk Surfaces in Ethereum.*** Cryptographic primitives within OpenZeppelin, despite their foundational role in enforcing trust, constitute disproportionately high-risk surfaces specifically in the Ethereum ecosystem due to their inherent complexity and pervasive, often uncritical, reuse. Critical functions like `processMultiProof`, observed in 1,407 Ethereum contract instances—fail to adequately validate Merkle tree integrity within NFT whitelisting mechanisms, enabling unauthorized asset minting through proof injection. Similarly, the `recover` function (identified in 843 high-risk Ethereum cases) is vulnerable to signature spoofing without stringent input validation, facilitating transaction replays and forged approvals. These instances demonstrate how subtle flaws in core cryptographic utilities can escalate into systemic attack vectors on Ethereum.

(*ii*) **Feature Updates Introduce Instability.** Feature updates to OpenZeppelin libraries prioritize new functionality and optimizations at the expense of backward compatibility and robust integration, introducing latent instability. This is starkly illustrated by Version 4.9.3, which exhibits a disproportionate concentration of security flaws, suggesting that rapid development cycles compromise stability testing and seamless interoperability with deployed contracts. Consequently, developers face heightened risks of introducing regressions or vulnerabilities when upgrading, forcing a trade-off between leveraging innovations and maintaining system reliability.

(*iii*) **Legacy Chains Amplify Risks.** The risks are further amplified on legacy chains, i.e., Ethereum, where technical debt and dependency inertia create uniquely compounded vulnerabilities. Contracts utilizing deprecated OpenZeppelin versions (e.g., <3.4) on Ethereum demonstrate significantly higher vulnerability density compared to those on BSC or Avalanche, largely due to immutable deployments, prohibitive migration costs, and entrenched code patterns. This persistence of outdated, vulnerable logic transforms Ethereum into a high-value, persistent attack surface resistant.

To further investigate OpenZeppelin-related vulnerabilities in real-world deployments, we prioritized critical-severity issues and conducted manual audits of high-value contracts. This process identified 16 exploitable vulnerabilities affecting contracts holding over $765,031 in cumulative assets. Findings were validated through cross-verification by the authors and an industry partner. We illustrate one such vulnerability through the following case study, which highlights systemic risks in widely deployed code.

Ethics: Any newly identified vulnerabilities were verified exclusively in test environments by the authors and our industry partner, with no public disclosure of these findings. A key challenge in largely anonymous blockchain networks, such as Ethereum, is the inability to directly contact the owners of vulnerable contracts [49]. To mitigate the risk of exploitation, we have applied additional code obfuscation to the provided snippets and intentionally omitted the names of the functions and details in the code.

Case Study. As shown in Fig. 4, this vulnerability originates from another issue [50] in the `ECDSA` contract of OpenZeppelin libraries (version ≤ 4.7.1). It arises from the functions accepting both traditional 65-byte signatures and the more compact 64-byte EIP-2098 signatures, which introduces signature malleability (lines 2 and 6). This means the same signature can be represented in multiple valid formats, allowing an attacker to convert a previously used signature into a different form and reuse it. Hence, attackers could bypass protections by submitting altered versions of reused signatures, leading to unauthorized actions like double spending.

VII. DISCUSSION

A. Threat to Validity

Internal Validity. The main threat to internal validity is the potential bias in our manual analysis of the security-related commit, the ground-truth dataset, and sample validation in the large-scale experiment. To mitigate this, we involved multiple authors in the analysis process with discrepancies resolved through discussion and consensus. Another potential threat is the reliance on the granularity of the line level and did not break changes down into smaller units. This limits our tool's ability to detect finer-grained changes, but it represents a trade-off between detection rate and false positive rate, aiming to prevent excessive false positives.

External Validity. The main external validity threat is the potential bias and not large enough in the large-scale experiment, which was conducted on the on-chain contracts of the Ethereum-compatible chains. In order to mitigate this threat, we selected the 3 different chains and the top 30,000 contracts from each chain. The total number of the contracts are comparable to the previous work [11].

B. Limitations and Future Work

ZEPCOMPARE's offline phase now relies on manual analysis to identify security-related commits and extract change facts. This process can be time-consuming and may not scale well with the increasing number of library versions. Automating this process using advanced techniques, such as machine learning or natural language processing, could enhance efficiency and scalability. Future work could explore these avenues to reduce manual effort and improve the tool's adaptability to new library versions.

ZEPCOMPARE encounters some challenges with specific patterns, mainly related to matching errors—that is, cases where functions are not correctly matched with the facts. As we discussed in RQ2 and RQ3, although our method extends function signatures to improve matching, there are still some corner cases that escape detection. Handling these cases would require more flexible and precise semantic analysis. However, using large language models (LLMs) for this purpose is both costly in terms of money and speed. To address this in future iterations, we will explore (*i*) optimizing the prompts and reducing the number of calls to LLMs, or (*ii*) collecting more data and designing lightweight, learning-based models tailored for the matching task, which could improve matching accuracy while keeping costs manageable.

VIII. RELATED WORK

Static Smart Contract Analysis for Vulnerability Detection. There are many prior works on smart contract static analysis for vulnerability detection. Some studies focused on rule-based methods to detect general vulnerabilities, such as Slither [18], Securify [51], SmartCheck [19], and Vanda [52]. Other studies employed techniques like symbolic execution, taint analysis, and model checking, including Manticore [21], Mythril [20], Osiris [53], Oyente [54], Ethainter [53], Zeus [46], Halmos [55], VetSc [56], and Pyrometer [57].

Some research also focused on specific vulnerabilities. For example, Liu *et al.* [58] presented SPCon, a tool based on role mining to detect access control vulnerabilities. Similarly, SOMO [59] and AChecker [60] used data flow analysis and symbolic execution for AC vulnerability detection. Cecchetti *et al.* [61] introduced a security condition for preserving key invariants while supporting secure reentrancy patterns. Sun *et al.* [62] proposed GPTScan, which connects LLM with static analysis to detect logic vulnerabilities. Unlike the above work, our research focuses on vulnerabilities in the OpenZeppelin code itself, conducting a systematic analysis and evaluating real-world smart contracts.

In particular, Liu *et al.* [11] proposed ZepScope, which analyzed whether the relevant OpenZeppelin functions of real-world contracts faithfully enforced the security checks in the official OpenZeppelin libraries. There are three key differences compared to our work: Conceptually, ZepScope assumes OpenZeppelin libraries are standardized and free of vulnerabilities, targeting only misuse of OpenZeppelin code, whereas we target vulnerabilities in the OpenZeppelin code itself. Technically, we compare candidate code segments with both vulnerable and fixed versions using *facts of changes* (see Section II-C and Section IV-B), while ZepScope mines only correct-version facts from the latest OpenZeppelin libraries. In detail, ZepScope focuses only on checking-based facts (i.e., `require` and `if-revert` statements), whereas we analyze all security-related statements as described in Section IV-B.

Clone Detection in Smart Contracts. Studies have found that code clones are more common in smart contracts than in other ecosystems. For example, Khan *et al.* [63] conducted a large-scale study on Ethereum, finding that about 79.2% of the code was cloned. Consequently, some research has used code clones to detect vulnerabilities in smart contracts. For example, Liu *et al.* [15] proposed a semantic-preserving representation for smart contract bytecode to search for vulnerabilities. Sun *et al.* [16] conducted a large scale study on the Ethereum blockchain to uncover its compositions, conduct code reuse analysis, and identify prevalent development patterns. However, these works focus only on clone detection and ignore the functional changes in smart contracts. Our research, in contrast, focuses on the widespread OpenZeppelin library and analyzes its change history to detect related vulnerabilities in various real-world smart contracts.

IX. CONCLUSION

This paper presents the first comprehensive analysis of vulnerabilities in OpenZeppelin libraries and their propagation to dependent smart contracts. We propose ZEPCOMPARE, an end-to-end system that analyzes flaws inherited from OpenZeppelin codebases by synthesizing facts of changes, a structured representation of vulnerability evolution across 64 library versions. Our study reveals that even when developers use OpenZeppelin correctly, 5.3% of contracts across three major chains inadvertently incorporate its latent vulnerabilities, spanning 20 categories from access control to cryptographic flaws. These findings demonstrate the risks emerge from flaws in foundational components themselves, underscoring the urgent need for proactive.

ACKNOWLEDGEMENTS

We thank all reviewers for their constructive comments. This research is partially supported by a research fund provided by HSBC, HKUST TLIP Grant FF612, and Lingnan Grant SUG-002/2526. This research is also supported by the National Research Foundation, Singapore, and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG4-GC-2023-008-1B); by the National Research Foundation Singapore and the Cyber Security Agency under the National Cybersecurity R&D Programme (NCRP25-P04-TAICeN); and by the Prime Minister's Office, Singapore under the Campus for Research Excellence and Technological Enterprise (CREATE) Programme. Any opinions, findings and conclusions, or recommendations expressed in these materials are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore, Cyber Security Agency of Singapore, Singapore.

References

[1] "Ethereum whitepaper," https://ethereum.org/whitepaper, 2024.

[2] D. Das, P. Bose, N. Ruaro, C. Kruegel, and G. Vigna, "Understanding security issues in the nft ecosystem," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2022, p. 667–681.

[3] S. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. Knottenbelt, "Sok: Decentralized finance (defi)," in *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, New York, NY, USA, 2023, p. 30–46.

[4] DeFiLlama, "Total value hacked," https://defillama.com/hacks, Nov 2024.

[5] "Openzeppelin library," https://github.com/OpenZeppelin/openzeppelin-contracts, Nov 2024.

[6] "Openzeppelin library - math," https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/math/Math.sol, Nov 2024.

[7] "Openzeppelin library - transparentupgradeableproxy," https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/proxy/transparent/TransparentUpgradeableProxy.sol, Nov 2024.

[8] "Openzeppelin library - access control," https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/AccessControl.sol, Nov 2024.

[9] Ethereum, "Erc-20 token standard," https://ethereum.org/en/developers/docs/standards/tokens/erc-20/, Nov 2024.

[10] X. Yi, Y. Fang, D. Wu, and L. Jiang, "BlockScope: Detecting and Investigating Propagated Vulnerabilities in Forked Blockchain Projects," in *Proceedings of the 2023 Network and Distributed System Security Symposium (NDSS)*, 2023.

[11] H. Liu, D. Wu, Y. Sun, H. Wang, K. Li, Y. Liu, and Y. Chen, "Using my functions should follow my checks: Understanding and detecting insecure OpenZeppelin code in smart contracts," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 3585–3601. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/liu-han

[12] "Solidity documentation - security considerations," https://docs.soliditylang.org/en/develop/security-considerations.html#use-the-checks-effects-interactions-pattern, Nov 2024.

[13] "OpenZeppelin Library - Security Advisories," https://github.com/OpenZeppelin/openzeppelin-contracts/security, Nov 2024.

[14] OpenZeppelin, "authorizeupgrade function," https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/proxy/utils/UUPSUpgradeable.sol, May 2025.

[15] H. Liu, Z. Yang, Y. Jiang, W. Zhao, and J. Sun, "Enabling clone detection for ethereum via smart contract birthmarks," in *Proceedings of the 27th International Conference on Program Comprehension*, 2019, pp. 105–115.

[16] K. Sun, Z. Xu, C. Liu, K. Li, and Y. Liu, "Demystifying the composition and code reuse in solidity smart contracts," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 796–807.

[17] "Smart contract weakness classification," https://swcregistry.io/, Nov 2024.

[18] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *Proceedings of the 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.

[19] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*, 2018, pp. 9–16.

[20] "Mythril," https://github.com/Consensys/mythril, Nov 2024.

[21] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Greico, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," Nov 2024. [Online]. Available: https://github.com/trailofbits/manticore

[22] "Website of ZepCompare," https://github.com/HelayLiu/ZepCompare, 2025.

[23] Ethereum, "Erc-721 non-fungible token standard," https://ethereum.org/en/developers/docs/standards/tokens/erc-721/, Nov 2024.

[24] "Cve-2021-39167," https://nvd.nist.gov/vuln/detail/CVE-2021-39167, Nov 2024.

[25] DeFiHackLabs, "Defi hacks," https://github.com/SunWeb3Sec/DeFiHackLabs, Nov 2024.

[26] OpenZeppelin, "Signature malleability vulnerability in 2.1.3 verion of openzeppelin library," https://github.com/OpenZeppelin/openzeppelin-contracts/pull/1622, Nov 2024.

[27] D. Lab, "Signature malleability vulnerability in tchtoken," https://web3sec.notion.site/c582b99cd7a84be48d972ca2126a2a1f?v=4671590619bd4b2ab16a15256e4fbba1&p=1ebfede1e2da486b89b94903f88c387e&pm=s, Nov 2024.

[28] "Pancakeswap Pairs," https://pancakeswap.finance/info/pairs, Nov 2024.

[29] "Openzeppelin Grant," https://blog.openzeppelin.com/openzeppelin-appointed-to-review-compounds-grant-proposals-to-improve-dao-security, Nov 2024.

[30] Y. Zhou, J. K. Siow, C. Wang, S. Liu, and Y. Liu, "Spi: Automated identification of security patches via commits," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 1, Sep. 2021.

[31] T. G. Nguyen, T. Le-Cong, H. J. Kang, X.-B. D. Le, and D. Lo, "Vulcurator: a vulnerability-fixing commit detector," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1726–1730.

[32] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 426–437.

[33] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, "Finding a needle in a haystack: automated mining of silent vulnerability fixes," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '21. IEEE Press, 2022, p. 705–716.

[34] T. Mikolov, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[35] D. S. Hirschberg, "Algorithms for the longest common subsequence problem," *Journal of the ACM (JACM)*, vol. 24, no. 4, pp. 664–675, 1977.

[36] J. R. Wood and L. E. Wood, "Card sorting: current practices and beyond," *Journal of Usability Studies*, vol. 4, no. 1, pp. 1–6, 2008.

[37] "Smart contract weakness classification - scw-122," https://swcregistry.io/docs/SWC-122/, Nov 2024.

[38] "Smart contract weakness classification - scw-117," https://swcregistry.io/docs/SWC-117/, Nov 2024.

[39] OpenZeppelin, "Ecdsa signature malleability in openzeppelin code," https://github.com/OpenZeppelin/openzeppelin-contracts/security/advisories/GHSA-4h98-2769-gh6h, Nov 2024.

[40] K. Li, Y. Xue, S. Chen, H. Liu, K. Sun, M. Hu, H. Wang, Y. Liu, and Y. Chen, "Static application security testing (sast) tools for smart contracts: How far are we?" *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024.

[41] Code4rena, "Code4rena audit," https://github.com/code-423n4, Nov 2024.

[42] Sherlock, "Sherlock Audit," https://github.com/sherlock-audit, Nov 2024.

[43] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the 2020 ACM/IEEE International Conference on Software Engineering (ICSE)*, 2020, pp. 530–541.

[44] Z. Zhang, B. Zhang, W. Xu, and Z. Lin, "Demystifying exploitable bugs in smart contracts," in *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 615–627.

[45] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "Sailfish: Vetting smart contract state-inconsistency bugs in seconds," in *Proceedings of the 2022 IEEE Symposium on Security and Privacy (IEEE SP)*. IEEE, 2022, pp. 161–178.

[46] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2018.

[47] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, "ethor: Practical and provably sound static analysis of ethereum smart con-

tracts," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 621–640.

[48] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer, "Scalable and probabilistic leaderless bft consensus through metastability," *arXiv preprint arXiv:1906.08936*, 2019.

[49] J. Krupp and C. Rossow, "teEther: Gnawing at ethereum to automatically exploit smart contracts," in *Proceedings of the 27th Usenix Security Symposium*, Baltimore, MD, Aug. 2018, pp. 1317–1333.

[50] "Cve-2022-35961," https://nvd.nist.gov/vuln/detail/CVE-2022-35961, Nov 2024.

[51] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.

[52] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[53] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: a smart contract security analyzer for composite vulnerabilities," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 454–469.

[54] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th annual computer security applications conference*, New York, NY, USA, 2018, p. 664–676.

[55] "Symbolic bounded model checker for ethereum smart contracts," https://github.com/a16z/halmos, Nov 2024.

[56] Y. Duan, X. Zhao, Y. Pan, S. Li, M. Li, F. Xu, and M. Zhang, "Towards automated safety vetting of smart contracts in decentralized applications," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 921–935.

[57] "Pyrometer," https://github.com/nascentxyz/pyrometer, Nov 2024.

[58] Y. Liu, Y. Li, S.-W. Lin, and C. Artho, "Finding permission bugs in smart contracts with role mining," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, Virtual South Korea, Jul. 2022, p. 716–727.

[59] Y. Fang, D. Wu, X. Yi, S. Wang, Y. Chen, M. Chen, Y. Liu, and L. Jiang, "Beyond "protected" and "private": An empirical security analysis of custom function modifiers in smart contracts," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, NY, USA, 2023, p. 1157–1168.

[60] A. Ghaleb, J. Rubin, and K. Pattabiraman, "Achecker: Statically detecting smart contract access control vulnerabilities," in *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, Melbourne, Australia, May 2023, p. 945–956.

[61] E. Cecchetti, S. Yao, H. Ni, and A. C. Myers, "Compositional security for reentrant applications," in *Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1249–1267.

[62] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639117

[63] F. Khan, I. David, D. Varro, and S. McIntosh, "Code cloning in smart contracts on the ethereum platform: An extended replication study," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2006–2019, 2023.