Silence False Alarms: Identifying Anti-Reentrancy Patterns on Ethereum to Refine Smart Contract Reentrancy Detection

Qiyang Song[†‡], Heqing Huang[†*], Xiaoqi Jia[†‡*], Yuanbo Xie[†‡], Jiahao Cao[§]
[†]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[‡]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[§]Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China
{songqiyang, huangheqing, jiaxiaoqi}@iie.ac.cn, xieyuanbo23@mails.ucas.ac.cn, caojh2021@mail.tsinghua.edu.cn

***Abstract*—Reentrancy vulnerabilities in Ethereum smart contracts have caused significant financial losses, prompting the creation of several automated reentrancy detectors. However, these detectors frequently yield a high rate of false positives due to coarse detection rules, often misclassifying contracts protected by anti-reentrancy patterns as *vulnerable*. Thus, there is a critical need for the development of specialized automated tools to assist these detectors in accurately identifying anti-reentrancy patterns. While existing code analysis techniques show promise for this specific task, they still face significant challenges in recognizing anti-reentrancy patterns. These challenges are primarily due to the complex and varied features of anti-reentrancy patterns, compounded by insufficient prior knowledge about these features.**

This paper introduces AutoAR, an automated recognition system designed to explore and identify prevalent anti-reentrancy patterns in Ethereum contracts. AutoAR utilizes a specialized graph representation, RentPDG, combined with a data filtration approach, to effectively capture anti-reentrancy-related semantics from a large pool of contracts. Based on RentPDGs extracted from these contracts, AutoAR employs a recognition model that integrates a graph auto-encoder with a clustering technique, specifically tailored for precise anti-reentrancy pattern identification. Experimental results show AutoAR can assist existing detectors in identifying 12 prevalent anti-reentrancy patterns with 89% accuracy, and when integrated into the detection workflow, it significantly reduces false positives by over 85%.

I. Introduction

Blockchain technology has been attracting significant attention from both industry and academia. Its decentralized consensus mechanism serves as the foundation for various open and decentralized platforms [32], [6], [20], allowing for transactions and program executions without a central authority. Among these platforms, Ethereum [6] is particularly notable for introducing Solidity smart contracts to develop decentralized applications. However, despite these advancements, the security of smart contracts has became a critical concern after the notorious DAO attack [28], which exploited a reentrancy vulnerability in Ethereum, leading to substantial financial losses.

[*]The corresponding authors.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.230167
www.ndss-symposium.org

In response, both industry and academic experts have developed a range of automated detectors to identify reentrancy vulnerabilities in contracts. Although some tools have achieved notable recall rates, they are prone to producing a significant number of false positives (FPs), with over 90% of reported cases estimated to be FPs [11], [16], [36], [55]. This high percentage of FPs can lead to confusion and fatigue among security analysts. Additionally, prior research [51] indicates that the average investigation time for FPs is approximately 30 minutes per contract per analyst. Thus, reducing FPs is crucial for maintaining the efficiency of security operations. Recent studies [11], [51] suggest that many FPs in reentrancy detection primarily arise from coarse detection rules that incorrectly classify contracts safeguarded by *anti-reentrancy patterns* as *vulnerable*. To improve the efficacy of these detectors, it is crucial to develop an automated tool capable of accurately identifying anti-reentrancy patterns in various contracts.

To our knowledge, there is a lack of specialized automated recognition tools for anti-reentrancy patterns. Although recent advancements in code analysis—including various static/dynamic analysis [3], [7], [53] and deep code learning techniques [22], [44]—have shown promise in identifying specific code patterns, designing a tool tailored for recognizing anti-reentrancy patterns in Ethereum contracts still remains significant challenges:

Challenge #1. A straightforward approach for identifying anti-reentrancy patterns in Ethereum contracts could involve manually defining rules and employing static/dynamic analysis to detect these patterns. However, this rule-based approach requires *priori knowledge* of anti-reentrancy patterns and lacks the flexibility to accommodate newly emerging patterns. Additionally, our review of existing literature [19], [43], [51], [54] (see Table III) reveals there is limited knowledge available for recognizing anti-reentrancy patterns commonly used in practice, rendering this rule-based approach inadequate for comprehensive identification of anti-reentrancy patterns.

Challenge #2. To circumvent the limitations of priori knowledge, we could employ deep learning to explore and identify anti-reentrancy patterns in Ethereum contracts. Nonetheless, due to the absence of *ground truth labels*, this technique may struggle to accurately learn the key features of anti-reentrancy patterns. Additionally, given the diversity and complexity of code information in Ethereum contracts, there is a risk that this technique might inadvertently learn *irrelevant code information*. Therefore, it is critical to develop specific methodologies and data structures that can precisely capture anti-reentrancy-related code information within contracts.

In this paper, we introduce AutoAR, an automated system that can effectively explore and identify anti-reentrancy patterns in Ethereum contracts. AutoAR aims to enhance all existing reentrancy detectors by aiding in the identification of anti-reentrancy patterns across various contracts. The system comprises three core components: *a data filtration approach*, *a specialized graph representation named RentPDG*, and *a recognition model*. Both the filtration approach and RentPDG are tailored to extract anti-reentrancy-related code information from Ethereum contracts, enabling our recognition model to accurately learn the key logic of anti-reentrancy patterns.

The data filtration approach is initially employed to pinpoint potentially valuable materials for model learning, specifically targeting those contracts likely equipped with anti-reentrancy patterns. The selection strategy is guided by the observation: *Ethereum contracts prone to reentrancy typically incorporate anti-reentrancy patterns*. By leveraging common reentrancy characteristics, this method effectively identifies relevant contracts. After data filtration, we further utilize the RentPDG to extract critical code information related to anti-reentrancy. Our RentPDG, a specialized variant of the inter-procedural program dependency graph (PDG), captures rich code syntax and cross-function control/data dependencies.

Unlike the conventional PDG, the RentPDG is tailored to retain only anti-reentrancy components. Since anti-reentrancy patterns often enforce control and data dependency constraints around an external call, constructing a RentPDG essentially involves *including only those PDG nodes and edges related to an external call*. Intuitively, a straightforward construction method might use reverse depth-first search (DFS) from an external call to analyze node and edge reachability, incorporating all directly connected elements. However, this DFS-based method does not consider inter-procedural contexts, which can lead to the erroneous inclusion of nodes and edges from infeasible paths that do not actually reach the external call. To overcome this issue, we develop a *context-sensitive reachability analysis* approach that integrates a specific context-free language grammar with traditional adjacency-matrix-based reachability analysis. This enhanced approach ensures precise path feasibility checks and reachability analysis, thereby enabling accurate RentPDG construction.

Building upon RentPDGs extracted from contracts, our recognition model employs a graph auto-encoder combined with a clustering technique to effectively detect anti-reentrancy patterns. The graph auto-encoder, equipped with a heterogeneous convolutional and attentional pooling mechanism, is designed to capture anti-reentrancy semantics embedded within RentPDGs. Specifically, the convolutional mechanism addresses the complexity of various edge interactions, while the pooling mechanism focuses on selecting nodes crucial for anti-reentrancy semantics. Together, the two mechanisms enable our graph auto-encoder to produce high-quality graph embeddings. By clustering these graph-level embeddings, our model can abstract and learn the key logic of the inherent anti-reentrancy patterns. In the subsequent recognition phase, the centroids of these clusters serve as key indicators for identifying similar anti-reentrancy patterns within contracts.

We implement a prototype of AutoAR for identifying anti-reentrancy patterns in Ethereum contracts, comprising 4,000 lines of code in Python. To assess its effectiveness, we conduct extensive experiments using a dataset of 115,240 unique, real-world Ethereum contracts. The experiments demonstrate that AutoAR can learn 12 distinct anti-reentrancy patterns via clustering. For these patterns, we found that state-of-the-art reentrancy detectors [4], [8], [9], [12], [48], [47] have coarse detection logic that overlooks at least 8 of these patterns. Furthermore, our results show that AutoAR significantly enhances the performance of existing reentrancy detectors. Specifically, it enables them to recognize anti-reentrancy patterns with an average accuracy of 89%, and when integrated into the detection workflow, it substantially reduces FPs by over 85%.

In summary, we make the following contributions:

- We present AutoAR, an automated recognition system that can be integrated with existing reentrancy detectors for identifying prevalent anti-reentrancy patterns on Ethereum;
- We develop a data filtration methodology and a novel graph data representation RentPDG, which effectively captures anti-reentrancy-related code information within Ethereum smart contracts;
- We design a recognition model capable of learning and identifying the key logic of anti-reentrancy patterns in Ethereum smart contracts;
- We implement a prototype of AutoAR and conduct extensive experiments to evaluate its efficacy. Our results show that AutoAR can aid existing reentrancy detectors in accurately identifying 12 prevalent anti-reentrancy patterns, thereby substantially reducing FPs. To our knowledge, AutoAR newly explores 8 anti-reentrancy patterns (P4-5 and P7-12) that are not publicly discussed in prior work. These security patterns can serve as guidelines not only for designing new reentrancy detectors but also for developing more secure contracts.

II. Background

Solidity is designed for smart contract development. It is an object-oriented language with the following features:

- *Function Modifiers:* Used in function declarations, they enforce specific conditions before or after a function executes, with the placeholder _ representing the function body.
- *Built-in Variables:* Solidity offers variables for capturing transaction details like *msg.sender* (sender's address), *msg.value* (amount of Ether sent), and *address.balance* (an address's Ether balance).

Reentrancy Attacks on Ethereum typically exploit repeated external function calls to manipulate state variables before they are properly updated. Figure 1 illustrates the key insight of a reentrancy attack, which takes advantage of state-read operations that precede external calls and state-written operations. By repeatedly invoking Ether transfer operations, *i.e., ExCall.call.value(var_s)*, attackers can steal significant financial assets.

Checks-effects-interactions, recommended by Solidity officials [43], safeguard contracts with external calls by ensuring

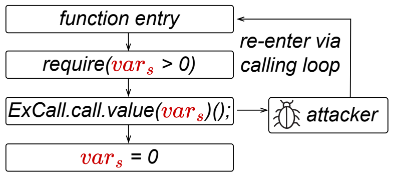


Fig. 1: Key Insight of a Typical Reentrancy Attack

all state-related checks, reads, and writes are completed before making external calls. This pattern preserves contract state consistency prior to external interactions, thus preventing state manipulation via re-entrant calls. Although this pattern is effective for preventing reentrancy, it is not always applicable, as some variables may depend on external call responses, and they should be written after the external calls.

Program Dependency Graph (PDG), an intermediate representation, combines control and data dependency graphs [13] for a detailed view of program syntax and semantics. When extended to an inter-procedural context, PDGs provide richer information, including cross-function control and data dependencies, along with calling context information.

III. System Overview

A. Problem Statement

Ethereum contract auditors frequently encounter a high rate of false positives (FPs) reported from reentrancy detectors due to the detectors' limited recognition of anti-reentrancy patterns. While most detectors identify the well-known checks-effects-interactions (CEI) pattern, they often miss other effective anti-reentrancy strategies. This paper aims to enhance the detectors' capability by identifying various anti-reentrancy patterns in Ethereum contracts, not limited to the CEI pattern.

Anti-reentrancy Pattern Recognition. To enhance the capabilities of existing reentrancy detectors, it is essential to recognize a broad spectrum of anti-reentrancy patterns. This includes both intentional anti-reentrancy code patterns and other patterns that, while not specifically designed to prevent reentrancy, can indirectly mitigate its risks. We aim to identify anti-reentrancy patterns with the following properties:

- *Property #1:* In most reentrancy events [2], [40], [37], attackers—typically unauthorized users—often exploit vulnerabilities via repeated function calls. Thus, an effective anti-reentrancy property is to restrict unauthorized users from arbitrarily re-entering sensitive functions.
- *Property #2:* Beyond simply preventing re-entrant function calls, another crucial anti-reentrancy property involves impeding attackers from manipulating contract states to gain unwarranted profits.

B. System Model and Workflow

Figure 2 illustrates the architecture of AutoAR, an automated recognition system that can *learn and identify* prevalent anti-reentrancy patterns within Ethereum contracts. This system can be used to enhance all state-of-the-art reentrancy detectors by significantly reducing FPs that arise from ignorance of anti-reentrancy patterns. Overall, AutoAR's pipeline consists of three main components: (i) *data filtration*, (ii) *anti-reentrancy related graph construction*, and (iii) *an anti-reentrancy recognition model.*

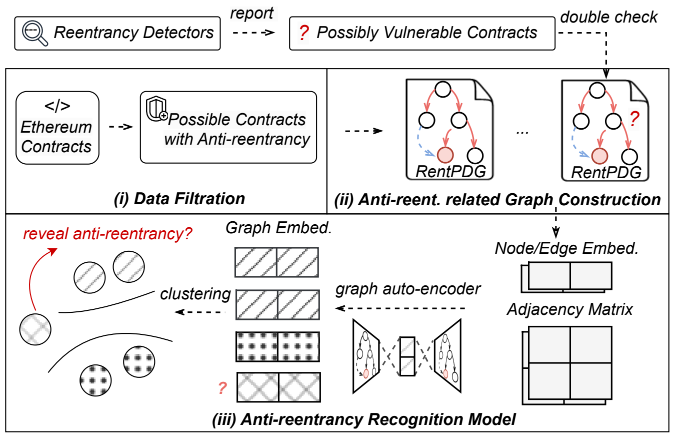


Fig. 2: AutoAR System

AutoAR operates in two phases: *training* and *recognition*. During training, AutoAR selects Ethereum contracts that are potentially enforced with anti-reentrancy patterns by (i), transforms the contracts to intermediate graph representations RentPDGs through (ii), and learns the characteristics of underlying anti-reentrancy patterns using (iii). In the recognition phase, AutoAR first converts the target contract to RentPDGs with (ii), and then determines if these RentPDGs reveal the presence of anti-reentrancy by (iii). Here, we briefly describe three components of AutoAR as follows.

Data Filtration. To accurately select datasets for learning anti-reentrancy patterns, we provide a data filtration approach aimed at identifying contracts likely enforced with anti-reentrancy patterns. The selection strategy is based on a key observation: *Ethereum contracts prone to reentrancy attacks are often enforced with anti-reentrancy patterns.* By leveraging established knowledge of reentrancy attacks, this approach can effectively select target contracts. Additionally, to enhance the precision of our filtration, we further provide a refinement approach to remove irrelevant contracts.

Anti-reentrancy Related Graph Construction. To precisely capture anti-reentrancy-related code information, we design a specialized program intermediate representation named RentPDG. It is a simplified version of inter-procedural program dependence graph (PDG) representation, including code syntax, cross-function control/data dependency, and inter-procedural calling context. Unlike traditional PDGs, our RentPDG only retains structures directly related to anti-reentrancy patterns. Given that anti-reentrancy patterns typically impose control and data dependency constraints on external calls, RentPDG construction is equivalent to extracting nodes and edges associated with external calls, namely, those that reach these calls.

A naive method for constructing RentPDGs involves using reverse deep first search (DFS) from external calls and then incorporating all the encountered nodes and edges. However, this method may mistakenly include nodes and edges from infeasible paths that do not actually connect to external calls, due to its disregard for inter-procedural context. To address this, we develop a *context-sensitive reachability analysis approach* that integrates context-free language grammar with traditional adjacency-matrix-based reachability analysis. This advanced approach can examine inter-procedural path feasibility and precisely analyze node and edge reachability, thereby enhancing the accuracy of RentPDG construction. For additional details on optimization approaches, please refer to Appendix B.

Anti-reentrancy Recognition Model. We develop an anti-reentrancy recognition model that integrates a graph auto-encoder with clustering to effectively learn anti-reentrancy patterns. Our graph auto-encoder, designed to *grasp anti-reentrancy semantics inherent in RentPDGs*, employs a heterogeneous graph mechanism with an attentional pooling mechanism. The advanced mechanisms handle diverse and complex edge interactions and identify the most critical nodes essential for recognizing anti-reentrancy patterns, respectively. By training on various RentPDGs and clustering the corresponding graph embeddings, our model can learn the fundamental logic of prevalent anti-reentrancy patterns on Ethereum. During the recognition phase, employing the cluster centroids with a detection threshold allows our model to detect if a contract's RentPDGs include anti-reentrancy patterns.

IV. Data Filtration

In this section, we propose a data filtration approach aimed at identifying contracts likely enforced with anti-reentrancy patterns. Considering that the Checks-Effects-Interactions pattern (CEI) is *well-recognized as non-vulnerable* by existing tools [4], [9], [12], [47], [48], we exclude CEI-guarded contracts from our dataset. We emphasize that this exclusion does not affect the performance of our system, as our goal is to help existing tools identify less-recognized anti-reentrancy patterns in contracts reported as *potentially vulnerable.*

To derive our final dataset step by step, we initially define Pattern 1, which represents a broad spectrum of Ethereum contracts prone to reentrancy. This dataset may be enforced with various anti-reentrancy patterns. Next, we introduce Pattern 2 to denote CEI-guarded contracts. By excluding Pattern 2 from Pattern 1, we have Pattern 3, which precisely captures our final dataset for learning.

A. Selecting Contracts Likely with Anti-reentrancy Patterns

Our data filtration method is based on the observation that Ethereum contracts prone to reentrancy attacks frequently employ anti-reentrancy patterns. While contracts not susceptible to reentrancy may also contain anti-reentrancy patterns, it is reasonable for our work to specifically focus on learning anti-reentrancy patterns within contracts prone to such attacks. This is coherent with our primary objective: assisting detectors in identifying anti-reentrancy patterns in contracts flagged as *potentially reentrancy-susceptible.* Thus, the essence of data filtration is to pinpoint reentrancy-susceptible contracts.

Identifying Reentrancy-Susceptible Contracts. Recall that a typical reentrancy-susceptible contract requires two conditions (§II): (i) an external call coexisting with state read and write operations; and (ii) a state read preceding an external call in the control flow. Accordingly, we define the following pattern to describe these contracts.

***Pattern** 1 (Reentrancy-Susceptible Contract):* A contract is deemed reentrancy-susceptible if any inter-procedural control flow path P contains:

$$\exists v_s \exists c\left((WR(v_s) \in P) \wedge (RD(v_s) \succ c)\right) \tag{1}$$

where v_s is a state variable stored in the persistent storage, c denotes an external call, with $WR(\cdot)$ and $RD(\cdot)$ representing write and read operations, and $\succ$ symbolizes sequence.

To identify the above pattern, we employ static analysis to construct a control flow graph and then examine state read/write operations in the context of paths related to external calls. While a few static analysis tools [12], [48], [9] offer mechanisms to detect such contracts, they fail to fully account for complex inter-procedural reentrancy attacks that can manipulate cross-function state variables through re-entrant calls. To better identify contracts prone to this attack type, we incorporate inter-procedural context into control flow graphs and extend our analysis to *inter-procedural scope*.

The inter-procedural analysis encompasses not only the context of function calls within a contract, but also extends to typical API calls of external contracts that can read and modify the current contract's state variables. By integrating these varied inter-procedural contexts, our approach ensures a thorough identification of reentrancy-susceptible patterns. Notably, this method could become time-consuming when a large number of inter-procedural paths exist. To mitigate this issue, we adopt a more efficient algorithm that simplifies this analysis by cutting inter-procedural paths and merging sub-paths (see more details in Appendix A).

Refinement. Initial filtration, based on Pattern 1, may inadvertently include some truly vulnerable contracts without anti-reentrancy patterns. To refine our dataset, we consult historical reentrancy attack data [37], which records 60 vulnerable smart contracts ever suffering from reentrancy attacks. We then exclude these contracts from our dataset. While prior work [11], [38] flags more vulnerable contracts, most are merely false positives, safeguarded by anti-reentrancy patterns. To ensure no potential anti-reentrancy patterns are missed, we limit the exclusion to the confirmed 60 contracts.

This refinement improves the precision of our dataset by focusing on non-vulnerable contracts likely to contain anti-reentrancy patterns. While this method may overlook some vulnerable contracts, which could remain in our dataset, it is still deemed reasonable since previous research [11], [51] suggests that genuinely vulnerable contracts constitute less than 0.2% of Ethereum contracts. This is coherent with the Ethereum reentrancy history data: only a few out of millions of contracts have been compromised [37]. To further mitigate the adverse effect of the contracts, we employ noise-resilient unsupervised learning to detect anti-reentrancy patterns (detailed in §VI-B).

B. Excluding CEI-guarded Contracts

Notably, our goal is to identify less-recognized anti-reentrancy patterns. We thus remove CEI-guarded contracts from our dataset as the CEI pattern is widely recognized by existing tools [4], [12], [47], [48]. The exclusion narrows our recognition scope and also reduces analysis noises.

CEI-guarded Contracts. The CEI requires specific sequences: *all pairs of read and write operations for reentrancy-susceptible variables should precede before external calls.*

Here, reentrancy-susceptible variables refer to those that could be manipulated in reentrancy attacks. Accordingly, we define the following pattern to characterize CEI-guarded contracts.

***Pattern** 2 (CEI-guarded Contract):* A contract is guarded by CEI if any inter-procedural control flow path P contains:

$$\forall v_s(Reentry(v_s) \rightarrow (RD(v_s) \succ c) \wedge (WR(v_s) \succ c)) \quad (2)$$

where v_s is a state variable stored in the persistent storage, c refers to an external call, with $RD(\cdot)$ and $WR(\cdot)$ denoting read and write operations, and $\succ$ indicating sequence. Particularly, $Reentry(v_s)$ denotes reentrancy-susceptible variables, which can be represented as follows:

$$Reentry(v_s) = \exists c((RD(v_s) \succ c) \wedge (WR(v_s) \in P)) \quad (3)$$

Identifying Final Dataset. Our final dataset excludes contracts with the CEI pattern. Therefore, the negation of Pattern 2 defines our final dataset, represented by the following pattern.

***Pattern** 3 (Final Dataset):* A contract belongs to the final dataset if any inter-procedural control flow path P contains:

$$\exists v_s \exists c((RD(v_s) \succ c) \wedge (c \succ WR(v_s)) \quad (4)$$

where v_s is a state variable stored in the persistent storage, c an external call, with $RD(\cdot)$ and $WR(\cdot)$ indicating read and write operations, and $\succ$ indicating sequence.

With this pattern, we can conduct static analysis to identify our final dataset. Similar to Pattern 1, we recognize this pattern in an inter-procedural manner (see Appendix A).

V. Anti-Reentrancy Related Graph Construction

This section introduces a method for constructing a specialized graph representation RentPDG, designed to capture anti-reentrancy-related code information. For additional details on optimization methods, please see Appendix B.

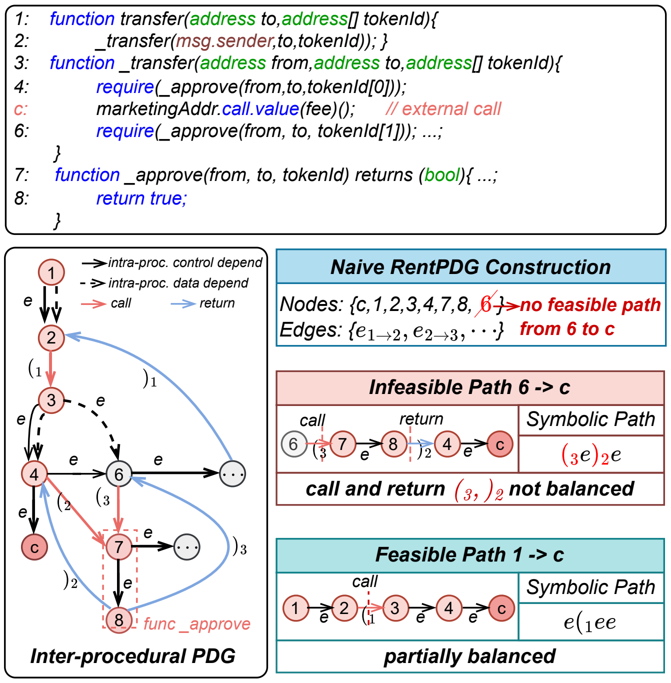
A. RentPDG Representation

We adopt a specialized variant of the inter-procedural program dependency graph (PDG), named RentPDG, to encompass extensive anti-reentrancy-related code information, including code syntax and cross-function control/data dependencies. The RentPDG also includes the inter-procedural calling contexts unique to Ethereum contracts, covering not only internal function calls within a contract but also typical API function calls provided by external contracts (see Appendix Table IV). Notably, the external API functions (*e.g., swapExactTokensForETH*) often interact with and trigger callbacks to invoking contracts, providing universally accessible services.

RentPDG. Unlike traditional PDG, our RentPDG focuses solely on graph components related to anti-reentrancy. As an anti-reentrancy pattern typically imposes control and data dependency constraints around an external call, RentPDG includes *only those PDG nodes and edges that connect to an external call*. It is denoted by the tuple *RentPDG (c) = (V, E)*, where V and E represent the sets of nodes and edges, respectively. The formulation is as follows:

$$\begin{aligned} V &= \{c\} \cup \{i \mid isReach_n(i, c)\} \\ E &= \{e_{i\rightarrow j} \mid isReach_e(e_{i\rightarrow j}, c)\} \end{aligned} \quad (5)$$

where c is an external call, i is a given node, and $e_{i\rightarrow j}$ denotes an edge from node i to j. The function $isReach_n(i, c)$ indicates the reachability from node i to c, and $isReach_e(e_{i\rightarrow j}, c)$ signifies the edge reachability from $e_{i\rightarrow j}$ to c.



```
1:  function transfer(address to,address[] tokenId){
2:          _transfer(msg.sender,to,tokenId)); }
3:  function _transfer(address from,address to,address[] tokenId){
4:          require(_approve(from,to,tokenId[0]));
c:          marketingAddr.call.value(fee)();      // external call
6:          require(_approve(from, to, tokenId[1])); ...;
      }
7:   function _approve(from, to, tokenId) returns (bool){ ...;
8:          return true;
      }
```

intra-proc. control depend
intra-proc. data depend
call
return
Inter-procedural PDG
func _approve
Naive RentPDG Construction
Nodes: {c,1,2,3,4,7,8, ~~6~~ } no feasible path from 6 to c
Edges: $\{e_{1\rightarrow 2}, e_{2\rightarrow 3}, \cdots\}$
Infeasible Path 6 -> c
call
return
Symbolic Path
$(_3 e)_2 e$
call and return $(_3,)_2$ not balanced
Feasible Path 1 -> c
call
Symbolic Path
$e(_1 ee$
partially balanced

Fig. 3: RentPDG Construction. Here, the lower left graph represents the original inter-procedural PDG of the contract. Based on this graph, naive RentPDG construction (blue box) may falsely include node 6 in an infeasible path (red box).

Naive RentPDG Construction. According to Equation 5, a RentPDG consists of nodes and edges that reach an external call node c. To construct a RentPDG, a naive approach involves generating an inter-procedural PDG, then performing a reverse depth-first search (DFS) from the external call c, and incorporating all encountered nodes and edges into the graph. This DFS-based approach, however, may mistakenly include nodes and edges from *infeasible paths* not actually reaching c, due to a lack of consideration of inter-procedural contexts.

Figure 3 illustrates an example. Initiating a reverse DFS from the external call node c would lead to the exploration of node 6. However, node 6 should not be included as it does not actually reach c. While there is a path from node 6 to c (i.e., $6 \rightarrow 7 \rightarrow 8 \rightarrow 4 \rightarrow c$), it contains a mismatched pair of call and return edges, $e_{6\rightarrow 7}$ and $e_{8\rightarrow 4}$, marking this path as infeasible. This is attributed to the fact that program execution cannot enter and exit a function at different locations, *i.e.,* nodes 6 and 4. Thus, constructing RentPDGs necessitates examining path feasibility in the inter-procedural context.

B. Inter-procedural Path Feasibility

To accurately extract nodes and edges from feasible paths, we follow weighted pushdown systems [41] to define a specific context-free language (CFL) for recognizing path feasibility. In our RentPDG, infeasible paths typically contain mismatched pairs of call and return edges. Thus, we employ a simplified

CFL that uses parentheses to represent the calling context: $(_i$ and $)_i$ for the call and return at the i-th call site, and the symbol e for other edges. This reduces the task of checking path feasibility to a balanced parenthesis problem.

Figure 3 illustrates an example. A path from node 6 to c can be symbolized as $(_3e)_2e$, with mismatched parentheses $(_3$ and $)_2$, indicating infeasibility. Notably, feasible symbolic paths may exhibit partial balance, with unbalanced opening parentheses in the prefix or unbalanced closing parentheses in the suffix. This is because they may not start and end in the same functions, showing only part of the calling context. For instance, the path $1 \xrightarrow{e} 2 \xrightarrow{(_1} 3 \xrightarrow{e} 4 \xrightarrow{e} c$, despite the unbalanced parenthesis $(_1$, is still feasible as the matching closing parenthesis $)_1$ appears later in execution.

The CFL grammar for path feasibility is defined as follows:

***Definition** 1 (Path Feasibility as CFL Grammar):* An inter-procedural path is feasible if it adheres to a CFL grammar $\{N, T, P, S\}$, defined by:

$$\begin{aligned} P: \quad & S \to U_1 \mid U_2 \\ & U_1 \to U_1 B \mid U_1 U_1 \mid \epsilon \mid (_i U_1 \\ & U_2 \to B U_2 \mid U_2 U_2 \mid \epsilon \mid U_2)_i \\ & B \to BB \mid e \mid \epsilon \mid (_i B)_i \end{aligned}$$

where N denotes non-terminal symbols $\{S, U_1, U_2, B\}$, T includes terminal symbols $\{e, (_1, \cdots\}$, P details the production rules, S is the staring symbol, and ϵ is an empty symbol. Particularly, B represents a parenthesis-balanced symbolic path, while U_1 and U_2 denote partially parenthesis-balanced symbolic paths: U_1 for those with unbalanced opening parentheses and U_2 for those with unbalanced closing parentheses.

Based on this definition, a classic stack-based parenthesis checker algorithm [39] can be used to examine path feasibility.

C. Context-sensitive Reachability Analysis for RentPDG

According to Equation 5, RentPDG construction requires the analysis of reachability from any given nodes and edges to an external call node. Given that infeasible inter-procedural paths may contribute to erroneous reachability information, we take inter-procedural path context into account and develop a context-sensitive reachability analysis approach.

Context-sensitive Reachability. Our analysis focuses on two types of context-sensitive reachability: (i) *context-sensitive node reachability* and (ii) *context-sensitive edge reachability*. Let $FPath(i, j)$ denote the set of all feasible paths from node i to node j. The two types of reachability are defined as follows:

***Definition** 2 (Context-Sensitive Node Reachability):* Context-sensitive node reachability from node i to j exists if:

$$isReach_n(i, j) = \exists p_{i \to j}(p_{i \to j} \in FPath(i, j)) \tag{6}$$

***Definition** 3 (Context-Sensitive Edge Reachability):* Context-sensitive edge reachability from edge $e_{i \to k}$ to node j exists if:

$$\begin{aligned} isReach_e(e_{i \to k}, j) = & \exists p_{k \to j}(p_{k \to j} \in FPath(k, j) \wedge \\ & \langle e_{i \to k}, p_{k \to j} \rangle \in FPath(i, j)) \end{aligned} \tag{7}$$

Here, $p_{k \to j}$ denotes a path from node i to j, and $\langle e_{i \to k}, p_{k \to j} \rangle$ symbolizes the concatenation of the edge $e_{i \to k}$ with the path $p_{k \to j}$, forming an extended path from node i to node j.

A conventional reachability analysis approach formulates the reachability problem as matrix multiplication [10], given an adjacency matrix $\mathbf{A}$. The h-th power of $\mathbf{A}$, denoted as $\mathbf{A^h}$, reveals h-order reachability information, namely the number of h-step paths between nodes. However, this method does not consider the inter-procedural calling context, potentially leading to inaccurate reachability results. To mitigate this, we integrate our CFL grammar into the matrix representation.

Symbolic Adjacency Matrix. The symbolic adjacency matrix $\mathbf{A_s}$ employs CFL symbols to represent various types of edges, enhancing the matrix's capacity to reflect the inter-procedural calling context. The elements of $\mathbf{A_s}$ are defined as follows:

$$\mathbf{A_s}[i][j] = \begin{cases} (_k, & \text{if } e_{i \to j} \text{ is a call edge} \\ )_k, & \text{if } e_{i \to j} \text{ is a return edge} \\ e, & \text{if } e_{i \to j} \text{ is a intra-procedural edge} \\ \emptyset, & \text{if there is no edge from i to j} \end{cases}$$

where $(_k$ and $)_k$ signify call and return edges at the k-th call site, respectively.

Symbolic Operations. In alignment with symbolic adjacency matrices, we introduce two fundamental symbolic operations: *symbolic multiplication* and *addition*. These operations facilitate complex symbolic matrix computation, including both standard and element-wise multiplication. By treating each matrix element as *a set of symbolic representations*, symbolic addition and multiplication correspond to the union and multiplication of two element sets, respectively. These operations are formally defined as:

$$\begin{aligned} E_1 + E_2 = \quad & E_1 \cup E_2 \\ E_1 \times E_2 = \quad & \{\langle p_a, p_b \rangle \mid p_a \in E_1, p_b \in E_2\} \end{aligned} \tag{8}$$

Here, E_1 and E_2 denote symbolic elements, with $\langle p_a, p_b \rangle$ denoting concatenation of their symbolic representations. Notably, the symbolic multiplication yields an empty set if either element is empty. This operation also accommodates interactions with binary numeric elements $\{0, 1\}$, allowing for operations such as $E_1 \times 1 = E_1$ and $E_1 \times 0 = \emptyset$.

Following prior work [10], [44], we can utilize symbolic matrix multiplication to evaluate context-sensitive node reachability. Nonetheless, this approach requires N^4 symbolic multiplication operations for N nodes, rendering it inefficient for large graphs. Note that RentPDG construction only requires the analysis of reachability information related to an external call node (Equation 5). Therefore, instead of applying symbolic matrix multiplication to analyze the reachability between all node pairs, we opt for a more lightweight approach: *symbolic matrix-vector multiplication*. This approach specifically targets context-sensitive node reachability towards an external call node. Additionally, we utilize *symbolic matrix element-wise multiplication* to derive context-sensitive edge reachability information related to the external call node.

Node Reachability Analysis. As illustrated in Figure 4, multiplying the symbolic adjacency matrix $\mathbf{A_s}$ with a specific numeric vector $\boldsymbol{\mu}_\mathbf{0}^\mathbf{c} = \{0, \cdots, 1\}$, where only the c-th element

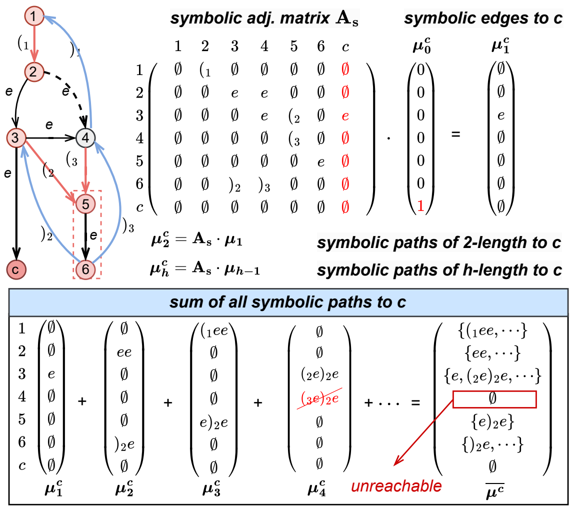

symbolic adj. matrix $\mathbf{A_s}$
symbolic edges to c
$\boldsymbol{\mu}_2^{\mathbf{c}} = \mathbf{A_s}\cdot\boldsymbol{\mu}_1$
$\boldsymbol{\mu}_h^{\mathbf{c}} = \mathbf{A_s}\cdot\boldsymbol{\mu}_{h-1}$
symbolic paths of 2-length to c
symbolic paths of h-length to c
sum of all symbolic paths to c
unreachable

Fig. 4: Context-sensitive Node Reachability Analysis

is 1, yields the first-order node reachability vector $\boldsymbol{\mu}_1^{\mathbf{c}}$. In this vector, the i-th element denotes the symbolic edge from node i to an external call c. Similarly, the h-order node reachability vector is computed as $\boldsymbol{\mu}_h^{\mathbf{c}} = \mathbf{A_s}\cdot\boldsymbol{\mu}_{h-1}^{\mathbf{c}}$, where the i-th element indicates symbolic paths of length h from node i to c. By compiling all reachability vectors and excluding infeasible paths, a context-sensitive node reachability vector is derived. This process is formulated as follows:

$$\overline{\boldsymbol{\mu}}^{\mathbf{c}} = \sum_{h=1}^{N} F_{fsb}(\boldsymbol{\mu}_h^{\mathbf{c}}), \quad \text{where } \boldsymbol{\mu}_h^{\mathbf{c}} = \mathbf{A_s}\cdot\boldsymbol{\mu}_{h-1}^{\mathbf{c}} \tag{9}$$

Here, N is the number of nodes. The function F_{fsb}, based on our CFL grammar (see Definition 1), serves to retain only those feasible paths in the final node reachability vector $\overline{\boldsymbol{\mu}}^{\mathbf{c}}$. Consequently, the final vector provides context-sensitive node reachability information. In particular, a non-empty $\overline{\boldsymbol{\mu}}^{\mathbf{c}}[i]$ signifies context-sensitive reachability from node i to node c.

Edge Reachability Analysis. To assess context-sensitive reachability from an edge $e_{i\to j}$ to node c, we conduct a *path combination trial*: appending the edge $e_{i\to j}$ to feasible paths from node j to c. If the combined path is feasible, it signifies that the edge can reach c. Notably, the path combination can be implemented by symbolic element-wise multiplication between the symbolic adjacency matrix $\mathbf{A_s}$ and node reachability vector $\overline{\boldsymbol{\mu}}^{\mathbf{c}}$. As Figure 5 shows, the operation $\mathbf{A_s}[4][5] \times \overline{\boldsymbol{\mu}}^{\mathbf{c}}[5]$ denotes the combination of edge $e_{4\to5}$ with symbolic paths from node 5 to c. Upon inspection, all combined paths are infeasible, indicating that c is unreachable from $e_{4\to5}$.

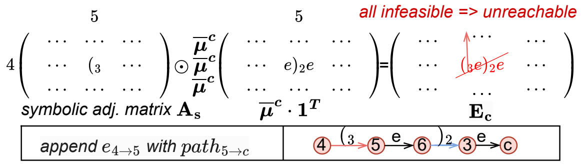

all infeasible => unreachable
symbolic adj. matrix $\mathbf{A_s}$
$\overline{\boldsymbol{\mu}}^{\mathbf{c}}\cdot\mathbf{1}^{\boldsymbol{T}}$
$\mathbf{E_c}$
append $e_{4\to5}$ with $path_{5\to c}$

Fig. 5: Context-sensitive Edge Reachability Analysis

To generalize this analysis to all edges, we employ symbolic matrix element-wise multiplication with $\mathbf{A_s}$ as follows:

$$\mathbf{E^c} = F_{fsb}(\mathbf{A_s} \odot (\overline{\boldsymbol{\mu}}^{\mathbf{c}}\cdot\mathbf{1}^{\boldsymbol{T}})) = F_{fsb}(\mathbf{A_s} \odot \begin{pmatrix} \overline{\boldsymbol{\mu}}^{\mathbf{c}} \\ \cdots \\ \overline{\boldsymbol{\mu}}^{\mathbf{c}} \end{pmatrix}) \tag{10}$$

Here, $\odot$ denotes symbolic matrix element-wise multiplication. The operation $\overline{\boldsymbol{\mu}}^{\boldsymbol{c}}\cdot\mathbf{1}^{\boldsymbol{T}}$ effectively expands the $n\times1$ vector $\overline{\boldsymbol{\mu}}^{\boldsymbol{c}}$ across N rows to match the dimensionality of $\mathbf{A_s}$, enabling element-wise symbolic multiplication. The function F_{fsb}, based on our CFL grammar, filters and retains only feasible symbolic paths within the final matrix $\mathbf{E^c}$. Consequently, a non-empty element $\mathbf{E^c}[i][j]$ reveals context-sensitive reachability from the edge $e_{i\to j}$ to node c.

VI. Anti-reentrancy Recognition Model

A. Graph Auto-Encoder for Anti-reentrancy

To input RentPDGs into our graph auto-encoder, we initially embed their graph topology, edges, and nodes into numerical matrices.

Initial Embedding. Graph topology is naturally represented by a numeric adjacency matrix. Notably, RentPDG contains four edge types that represent different dependencies: intra-procedural and inter-procedural control dependencies, along with local and global data dependencies. Consequently, we use 4-dimensional one-hot vectors for edge embedding. For node embedding, we integrate syntactic and semantic attributes into a vector. The inclusion of syntactic attributes, akin to semantic ones, is crucial for recognizing anti-reentrancy patterns, as some patterns may employ similar code snippets.

Syntactic attributes of nodes are processed by treating code tokens as words and nodes as sentences or documents, which allows us to utilize natural language processing techniques. To minimize noise, we first apply code normalization to remove whitespaces and comments, and standardize constants and address literals/variables. After normalization, we use the default configuration of Genism document embedding model [21] to map the code into a 100-dimensional vector space.

Beyond the initial 100 dimensions, we incorporate additional dimensions to capture semantic attributes per node. As previously mentioned, anti-reentrancy patterns often impose control and data dependency constraints around external calls. Accordingly, we choose semantic attributes that reveal specific function calls, node control flow types, and characteristics of data operations (*i.e.,* arithmetic and logic operations). Considering the Solidity community may provide specific mechanisms to thwart reentrancy, we also select attributes related to common API calls. Based on these observations, we extract the following 60 attributes (detailed in Appendix C).

- *Variables and Literals (17):* thirteen attributes for constants, variable numbers, and types, as well as four attributes for specific Solidity variables.
- *Function Calls (6):* six attributes for revealing various function calls, including Ether transfer-involved functions.
- *Operators (4):* four attributes for the number of relational, logical, arithmetic, and bitwise operators.
- *Node Type (18):* eighteen attributes indicating specific control flow node types.
- *Built-in API calls (12):* one attribute for the count of built-in Ether transfer functions, plus eleven attributes for revealing built-in functions related to access control.
- *Library API calls (3):* an attribute for the count of library calls, with two attributes for revealing verification and context function calls.

General Model Design. Our graph auto-encoder is tailored to capture anti-reentrancy semantics inherent in RentPDGs. The model, depicted in Figure 6, comprises an encoder and a decoder network. Following the configurations of existing graph auto-encoders [45], [49], we set the encoder and decoder to only three layers deep to prevent graph oversmoothing. Specifically, the encoder employs three convolutional and pooling layers to process initial embeddings into latent node-level embeddings. Conversely, the decoder utilizes three convolutional and unpooling layers to reconstruct the original embeddings from these latent representations, thereby self-supervising the model's training process. Additionally, to encapsulate the semantics of the entire graph more effectively, we introduce a graph representation layer that aggregates latent node-level embeddings into a unified graph-level embedding.

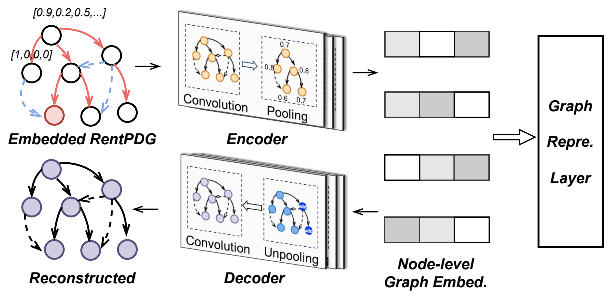


Fig. 6: Graph Auto-encoder for Anti-reentrancy

Heterogeneous Graph Convolution. Due to the diverse edge types in RentPDG, we utilize a heterogeneous graph convolutional mechanism, which is designed to incorporate neighbor node information via different edge types. To achieve this, we assign different weight parameters to each edge type. For a node i, the convolutional process is represented as:

$$\boldsymbol{f_{h+1}^i} = \mathbf{W_0}\boldsymbol{f_h^i} + \frac{1}{4|\mathcal{N}(i)|}\sum_{j\in\mathcal{N}(i)}\sum_{k=1}^{4}\boldsymbol{e_{j\to i}}[k]\cdot\mathbf{W_k}\boldsymbol{f_h^j} \quad (11)$$

where $\boldsymbol{f_h^j}$ refer to node j's feature vector at layer h, $\mathcal{N}(i)$ denotes node i's neighbors, and $\boldsymbol{e_{j\to i}}$ is the edge embedding from node j to i. The k-th element of $\boldsymbol{e_{j\to i}}$ is a binary variable indicating if the edge belongs to the k-th type. $\boldsymbol{W_0}$ and $\boldsymbol{W_k}$ denotes the convolutional weight parameters for the central node i and the connected edge of the k-th type, respectively.

Graph Attentional Pooling. In RentPDGs, different nodes contribute variably to the semantics of anti-reentrancy patterns. To effectively capture the crucial nodes, we introduce a graph attentional pooling mechanism. This mechanism calculates node contribution scores using the following formula:

$$s_i = Norm(\sum_{j\in\mathcal{N}(i)}\boldsymbol{f_i^T}\cdot\boldsymbol{f_j}) \quad (12)$$

where s_i is node i's contribution score, $\mathcal{N}(i)$ denotes neighbor nodes, $\boldsymbol{f_j}$ is node j's feature vector, and $Norm(\cdot)$ normalizes scores to $[0,1]$. Based on these scores, the pooling mechanism sets a threshold pooling ratio p to retain only the top p% of crucial nodes. Notably, this pooling mechanism may inadvertently remove an external call node, which is vital for revealing anti-reentrancy semantics. Thus, we assign a score of 1 to any call node, ensuring it remains during the pooling process.

Graph Unpooling. The unpooling mechanism serves as the counterpart to the pooling mechanism, aiding the decoder network in reconstructing the original graphs, thereby facilitating self-supervised training of the entire model. Specifically, each unpooling layer reintroduces nodes and edges based on the topology information recorded in the previous pooling layers.

Graph Representation Layer. This layer synthesizes latent node embeddings from the encoder into a unified graph-level embedding $\boldsymbol{f_G}$, representing the overall graph's semantics:

$$\boldsymbol{f_G} = \langle\boldsymbol{f_c}, mean_{j\in G}\boldsymbol{f_j}\rangle \quad (13)$$

where $\boldsymbol{f_c}$ denotes the embedding of an external call c, and $mean_{j\in G}\boldsymbol{f_j}$ refers to average node embedding.

B. Clustering-based Recognition

Graph-level embeddings from our graph auto-encoder capture the anti-reentrancy semantics inherent in RentPDGs. These embeddings are expected to naturally cluster based on their distinct anti-reentrancy characteristics, facilitating unsupervised anti-reentrancy pattern identification. Here, we employ the Genieclust hierarchical clustering algorithm, known for its robustness against noisy data points and effectiveness in handling non-convex datasets [14]. This algorithm requires the pre-configuration of cluster numbers. To determine the optimal configuration, we use the silhouette coefficient metric, a common measure in unsupervised learning for evaluating clustering quality. Generally, the configuration with the highest average silhouette coefficient is deemed most appropriate.

Training and Recognition. To train our model, we initially select contracts likely employing anti-reentrancy patterns from a diverse array of Ethereum contracts. These contracts are then transformed into RentPDGs, from which we generate graph-level embeddings. Using the Genieclust algorithm, we cluster these embeddings, where the centroids of these clusters serve as key indicators for detecting anti-reentrancy patterns during the recognition phase. Specifically, we define a detection threshold τ, representing a deviation that exceeds the mean distance from each cluster's centroid. RentPDG embeddings that fall outside this threshold are classified as outliers, indicating the probable absence of anti-reentrancy patterns.

Integration with Reentrancy Detectors. Our trained model can be seamlessly integrated into the workflow of existing detectors to precisely identify anti-reentrancy patterns and reduce false positives. When a contract is flagged as *potentially vulnerable*, our model double-checks their RentPDGs. If all are found to reveal anti-reentrancy patterns, the contract is deemed *not vulnerable*, and is thus removed from the suspect list.

VII. Experimental Evaluation

We evaluate AutoAR with an emphasis on answering the following research questions:

RQ1: What anti-reentrancy patterns can AutoAR reliably explore and learn within Ethereum contracts?(§VII-B)

RQ2: How many anti-reentrancy patterns are typically overlooked by existing reentrancy detectors? (§VII-C)

RQ3: How effectively can AutoAR help existing detectors identify anti-reentrancy patterns? To what extent does it enhance their detection performance? (§VII-D)

RQ4: What is AutoAR's computational overhead? (§VII-E)
RQ5: How important are the different components in our RentPDG construction module? (§VII-F)

A. Experimental Setup

Our experiments are performed on a PC equipped with Ubuntu 20.04, featuring 16 Intel Xeon 3.60GHz processors, 16GB RAM, and an NVIDIA T4 GPU.

Implementation. The prototype of AutoAR, consisting of approximately 4,000 lines of code (LoC) in Python, utilizes the Slither lexical analyzer [12] to construct interprocedural program dependence graphs. Additionally, it employs Python scripts for context-sensitive reachability analysis. Initial graph embedding is facilitated using the Slither and Gensim Word2vec libraries [15]. The graph auto-encoder is developed with the Pytorch-Geometric 1.6 library. We set the learning rate at 0.01 and the batch size to 256.

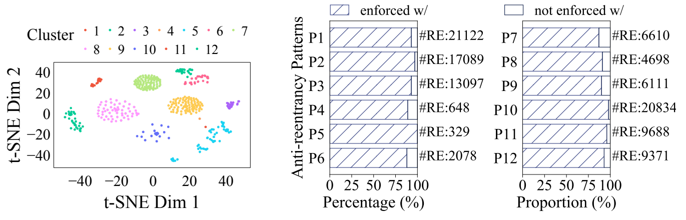
Dataset. Real-world Ethereum smart contracts are extracted from Etherscan, which offers an API to download source code by contract address. We query 40,000 addresses with available source code deployed from June 1, 2022, to May 13, 2023, and retrieve 190,558 Solidity files. After removing duplicates, we refine the data to 115,240 unique contracts. This dataset includes diverse contracts from Etherscan, such as ERC20, ERC721, ERC777, and ERC1155, as well as non-standard contracts with complex structures. On average, each contract contains 60 functions and 306 lines of code. Detailed statistics on contract structures are available in Appendix D. We divide the dataset into 80% for training and 20% for testing. From the training dataset of 92,192 contracts, AutoAR filters out 34,081 contracts and extracts 111,681 RentPDGs for learning.

System Parameters. To determine the optimal settings, we conduct several experiments with various parameter configurations to observe changes in clustering quality (see Appendix E). Finally, AutoAR is configured with a down-sampling ratio of 0.7, a pooling ratio of 0.8, and 12 clusters.

B. RQ1: Learned Anti-reentrancy Patterns

By clustering contracts' RentPDG embeddings, our AutoAR effectively learns and abstracts the features of prevalent anti-reentrancy patterns in Ethereum contracts. The clustering results are visually represented in Figure 7(a), where AutoAR produces 12 distinct clusters, each clearly separated from the others. To interpret these results, we randomly select 50 contracts corresponding to each cluster and analyze their shared patterns. Our analysis reveals a majority of contracts for each cluster exhibit a consistent anti-reentrancy pattern (P1-12). Even though a few contracts may differ, they are also secure and enforced with other patterns. Detailed statistics of clustering are presented in Figure 7(b), showing that over 86% of contracts for each cluster reveal a specific anti-reentrancy pattern. The patterns are detailed as follows.

P1: Safe Ether Transfer. Solidity built-in Ether transfer functions, *send()* and *transfer()*, are capped at a 2300 gas limit. This limit suffices for logging but not for complex operations, effectively preventing reentrant calls that typically require more gas.



(a) t-SNE

(b) Cluster Statistics. Here, #RE denotes the number of clustered contracts' RentPDG embeddings.

Fig. 7: Clustering Results after Training

P2: Mutex Variable. This pattern employs a mutex variable akin to conventional process mutex mechanisms. It functions by surrounding a specific code snippet to either lock or release it. The variable *externalCallFlag* is set to *true* before the external call, signifying the code is in a locked state. By adding a check for the lock status, it prevents attackers from re-entering the function through the call. After the call is completed, *externalCallFlag* is reset to *false*, indicating that the code is now in a released state. This mechanism effectively mitigates the risk of reentrancy attacks by controlling the control flow around the call. Typically, this pattern can also place the setting of mutex variables within a *function modifier* to protect specific functions during execution.

```
function request(uint8 requestType, bytes32[] requestData)
     public payable returns (int) {
    if(!externalCallFlag){//check for the lock status
        externalCallFlag = true; //lock
        //external call
        if (!msg.sender.call.value(msg.value)()) throw;
        externalCallFlag = false;//release
        /* some code omitted */}}
```

Listing 1: Mutex Variable

P3: Sender Check. This pattern often employs a function modifier to check the message sender's identity, typically using *require* or *if* statements to ascertain if the sender is within a whitelist or is an authorized user. For instance, a typical implementation might include a check *require(_owner == msg.sender)* within the function modifier, where *_owner* represents the contract owner's address. Such a check ensures that only the owner can access the function, effectively preventing reentrancy from unauthorized users.

```
function proxy(bytes[] calldata signs, uint256 nonce,
     address addr, bytes calldata input) external{
bytes32 hash = keccak256(abi.encodePacked(PROXY_USAGE, nonce
     , addr, input));
// signature validation
for(uint256 i = 0; i < signatures.length; i++) {
    address signer = hash.recover(signs[i]);//recover signer
    require(authorized[signer], "address is ..."); // check
    bool succ  = addr.call(input); // external cal
    /*some code omitted*/}}
```

Listing 2: Parameter Validation

P4: Parameter Validation. Compared to P3, this pattern focuses on validating function parameters rather than solely checking the sender's identity. It typically verifies if the parameters originate from a trusted sender, primarily through two approaches: *signature validation* and *Merkle proof validation*. Listing 2 showcases the signature validation approach, where

ECDSA.recover is used to recover the signer's address from *signs[i]*, followed by a check to see if the address is included in a whitelist. Notably, a whitelist is not always necessary. For example, the Merkle proof validation only requires maintaining a Merkle root for parameter validation.

P5: EOA Restriction. This pattern employs a specific check, *require(tx.origin == msg.sender)*, to prevent external contracts from invoking a function. The built-in Solidity variable *tx.origin* refers to an address of an externally owned account (EOA), which does *not contain any executable code*. Thus, ensuring that the caller is an EOA, rather than another contract, effectively blocks the execution of reentrancy attacks.

P6: Reentrancy Guard. This pattern is provided by OpenZeppelin [34], a widely used open-source contract framework. It employs a function modifier to prevent reentrancy in specific functions. Listing 3 illustrates this pattern. The modifier *nonReentrant* uses a status variable *_status* and a check (Line 4) to lock the *mint* function. While this pattern is similar to P2, it differs by encapsulating low-level operations and abstracting the locking logic within a modifier of an abstract contract. As a result, the protection is confined to the scope of a single function call, making it less perceptible but also less flexible in handling cross-function and cross-contract reentrancy scenarios.

```
1  abstract contract ReentrancyGuard{
2      uint256 private _status;
3      modifier nonReentrant() {
4          require(_status != _ENTERED, "...");//check
5          _status = _ENTERED; // status: entered
6          _;
7          _status = _NOT_ENTERED; // status: not entered
8      }}
9  function mint(uint256 _mintAmount) public payable
        nonReentrant {
10     /* some code omitted */
11     try IERC721Receiver(to).onERC721Received(_msgSender(),
          from, tokenId, _data {...}; //external call
12 }
```

Listing 3: Reentrancy Guard

P7: Access Frequency Limitation. This pattern leverages a state variable alongside a conditional check to control the frequency at which functions can be accessed. Listing 4 shows an example. Within the *autoBurnLPTokens* function, the variable *lastLpBurnTime* records the timestamp of the last token burn. A check is then implemented (Line 7) to ensure that external calls cannot re-enter the function within a short time frame, effectively limiting access frequency and reducing the potential for reentrancy.

```
1  function autoBurnLPTokens() internal returns (bool) {
2      lastLpBurnTime = block.timestamp; // last access time
3      pair.sync(); //external call
4  }
5  function _transfer(address from, address to, uint256 amount)
        internal override {
6      /* some code omitted */
7      if (block.timestamp > lastLpBurnTime + lpBurnFreq)
8          //check access frequency
9          autoBurnLPTokens();}
```

Listing 4: Access Frequency Limitation

P8: Access Price. This pattern places a financial barrier at the start of a function: *require(msg.value > cost * AMOUNT)*, where *msg.value* is the amount of Ether sent. By imposing this cost threshold, the pattern significantly raises the expense associated with potential reentrancy attacks, thereby deterring rational attackers from attempting to exploit the function.

P9: Post-reentrancy Check. This pattern involves placing checks on state variables after an external call to detect any state manipulations. While it cannot initially detect reentrancy, it indeed enables the detection of reentrancy attacks after the fact. As shown in Listing 5, if a state variable *_currentIndex* changes unexpectedly during a recursive function call, the state check at Line 5 can confirm that it loses consistency with its initial value, thereby effectively preventing reentrancy attacks.

```
1  function _mintTransfer(address to) internal {
2          uint256 startId = _currentIndex;
3          /**some code ommitted here**/
4          try IERC721Receiver(to).onERC721Received(_msgSender()
            , from, startId + 1, _data);//external call
5          if (_currentIndex != startId) revert(); //post-check
6          _currentIndex = startId + 1;}
```

Listing 5: Post-reentrancy Check

P10: Fixed Contract Address. This pattern fixes crucial external contract addresses during contract creation and restricts further modifications, allowing only the owner to make changes. This approach enhances security by preventing attackers from redirecting external calls to their controlled contracts.

P11: Intermediate State Update. This pattern requires developers to promptly update state variables after certain intermediate operations, rather than postponing updates until all operations are complete. As demonstrated in Listing 6, variables such as *tokensForLiquidity* and *tokensForMarketing* are updated twice (Lines 6 and 9). The first update, immediately following liquidity addition operations, serves as an intermediate state update, setting the state variables to 0. This strategy prevents attackers from re-entering the function via an external call, as the variable *totalTokens* will be 0 upon reentry, causing the check at Line 3 to fail. Unlike the checks-effects-interactions mechanism, which requires completing all state writes before external calls, this mechanism is more flexible. It is particularly suitable for scenarios where the final values of contract states cannot be predetermined before external calls.

```
1  function _transfer(address to, uint256 amount) internal {
2      uint256 totalTokens = tokensForLiquidity +
          tokensForMarketing;
3      if(totalTokens == 0) //check states
4          return;
5      /* liquidity addition operations omitted */
6      tokensForLiquidity = 0, tokensForMarketing = 0; //
          intermediate state update
7      (succ, ) = address(dev).call{value: ethDev}(""); //
          external call
8      if (autoMarketMakerPairs[to] && TotalFees > 0){
9          tokensForLiquidity = fee * LiquidFee / TotalFees,
              tokensForMarketing = fee * MarketFee /
              TotalFees;
10     }}
```

Listing 6: Intermediate State Updating

P12: State Sync from External Contracts. This pattern utilizes callbacks from external contracts to synchronize the state of the current contract, preventing manipulation through re-entrant calls. As illustrated in Listing 7, even if an attacker tries to initiate reentrant calls to transfer *eth* repeatedly, the external API call to the Uniswap function triggers a callback to the current contract's function, which immediately sets *_balances[this]* to zero. Consequently, the state check at Line 3 effectively prevents attackers from re-entering the function.

```
function _transferFrom(address from, address to, uint256
     amount) internal override {
    uint256 ctBalance = _balances[address(this)];
    if (ctBalance == 0) return; //check state
    //Uniswap API call: swap ctBalance tokens for Ether. It
         will trigger a callback to update '_balances'.
    uint256 initialBalance = address(this).balance;
    uniswapAPI.swapExactTokensForETH(ctBalance, 0, path,
         this, block.timestamp);
    uint256 eth = address(this).balance - initialBalance;
    address(wallet).call{value:eth}(""); //external call
    /* some code omitted*/}
```

Listing 7: State Sync from External Contracts

Remark. Except for P1, which imposes direct gas restrictions on external calls, all other anti-reentrancy patterns enforce restrictions on control and data dependencies related to external calls. Specifically, P2-9 typically implement control dependency checks around external calls, which are in turn dependent on the status of certain variables (data dependencies) to prevent reentrancy. Conversely, P10-12 generally impose data dependency restrictions on external addresses and state variables, complemented by control dependencies to safeguard the integrity of external addresses and ensure the consistency of state variables during reentrancy. Notably, our RentPDG can capture all these dependencies, allowing our system to accurately learn anti-reentrancy semantics.

C. RQ2: Anti-reentrancy Patterns Ignored by Existing Tools

To evaluate the efficacy of AutoAR in enhancing the detection capabilities of existing reentrancy detectors, we assess how many anti-reentrancy patterns, learned by AutoAR, are typically overlooked by state-of-the-art detectors, including Mythril [9], Slither [12], Securify [47], Conkas [48], Smartian [8], and Sailfish [4]. A direct assessment method involves scanning anti-reentrancy-guarded contracts to see if these detectors report any reentrancy risks. Intuitively, the absence of risk reports for the contracts with a specific anti-reentrancy pattern might suggest the detectors' ability to recognize this pattern. However, this approach has its limitations: a lack of risk reports does not conclusively demonstrate a detector's effective recognition of anti-reentrancy; rather, it may simply reveal the detector's rough detection rules, such as only checking intra-function patterns. Accordingly, a more reliable evaluation method is needed to accurately determine each detector's capabilities in recognizing anti-reentrancy patterns.

Before-and-after Comparison Experiment. To address these limitations, we adopt a more reliable method by comparing detector responses before and after anti-reentrancy patterns are integrated into those contracts known to be vulnerable. Specifically, if a detector flags a contract as *'vulnerable' before and 'not vulnerable' after the enforcement* of an anti-reentrancy pattern, it demonstrates that the detector can accurately recognize the anti-reentrancy pattern.

For this study, we select 31 contracts identified as reentrancy-vulnerable from SmartBugs [11] and develop Python scripts to embed 12 anti-reentrancy patterns into these contracts[1]. The modified contracts are then re-analyzed by the six detectors. Our experiment tracks whether each detector's response shifts from *vulnerable* to *not vulnerable* after the enforcement of each anti-reentrancy pattern, providing a precise measure of each detector's recognition capability.

[1]The anti-reentrancy injection scripts are available in https://github.com/ashessqy126/Anti-reentrancy-Pattern-Injection

TABLE I: Comparison Experiments. Here, 6 tools are applied to scan contracts before and after anti-reentrancy enforcement.

Setup		Slither	Securify	Mythril	Conkas	Smartian	Sailfish
Detection Round #1	Original*	31	29	10	31	13	28
Detection Round #2	w/ P1	**0/31**	29/29	10/10	31/31	**0/13**	**0/28**
	w/ P2	31/31	29/29	10/10	**4/31**	**0/13**	**2/28**
	w/ P3	31/31	29/29	10/10	31/31	13/13	28/28
	w/ P4	31/31	29/29	10/10	31/31	13/13	28/28
	w/ P5	31/31	29/29	10/10	31/31	13/13	28/28
	w/ P6	31/31	29/29	10/10	**4/31**	**0/13**	**2/28**
	w/ P7	31/31	29/29	10/10	31/31	13/13	28/28
	w/ P8	31/31	29/29	10/10	31/31	13/13	28/28
	w/ P9	31/31	29/29	10/10	31/31	13/13	28/28
	w/ P10	31/31	29/29	10/10	31/31	**0/13**	28/28
	w/ P11	31/31	29/29	10/10	31/31	13/13	28/28
	w/ P12	31/31	29/29	10/10	31/31	13/13	28/28

* It refers to original, vulnerable contracts without anti-reentrancy patterns enforced.

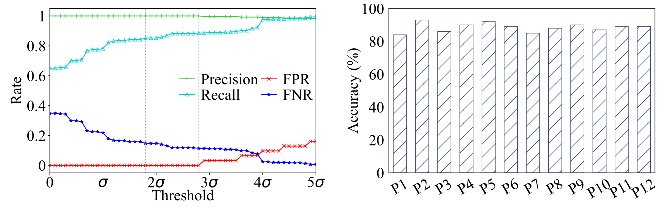
Table I presents the experimental results. Initially, we employ six detectors to scan the original, vulnerable contracts. We note that the detectors identify a varying number of reentrancy-vulnerable contracts. Among the six detectors, Slither and Conkas achieve the highest recall ratio (100%). For each detector, we record contracts marked as *vulnerable* and then use the corresponding anti-reentrancy-enforced versions for a second round of scanning. In the second round, we observe that all detectors fail to recognize at least 8 anti-reentrancy patterns. Particularly, Securify and Mythril flag all contracts enforced with P1-12 *vulnerable*, indicating that they lack any recognition logic for these 12 anti-reentrancy patterns.

Furthermore, we observe that Slither does not report any contracts with the P1 (*safe Ether transfer*) pattern, suggesting its potential effectiveness in identifying P1. Our review of its source code confirms that it indeed has rules to recognize safe Ether transfers. Additionally, Conkas, Smartian, and Sailfish report only a few cases out of 31 contracts with P2 (*mutex variable*) and P6 (*reentrancy guard*), indicating the presence of detection rules for these patterns. Our source code analysis shows their use of symbolic execution and fuzzing engines to recognize path constraints related to P2 and P6. Notably, Smartian does not report any contracts with P10 (*fixed contract address*), revealing its ability to identify P10. Further analysis of its fuzzing oracle shows it indeed classifies all paths initiated by internal fixed address as *non-vulnerable*.

D. RQ3: Efficacy of AutoAR's Anti-reentrancy Recognition

To evaluate how effectively AutoAR aids existing detectors in identifying contracts safeguarded by anti-reentrancy patterns, we initially employ six popular detectors—Slither, Securify, Mythril, Conkas, Smartian, and Sailfish—to scan 23,048 real-world testing contracts. These detectors flag 6,841 contracts as potentially vulnerable. Then, we randomly select

300 of these contracts and invest 5 hours in thorough inspections, finally confirming 298 (99%) as FPs; these are indeed safeguarded by various anti-reentrancy patterns. Subsequently, AutoAR is employed to recognize these contracts. To ensure a fair evaluation, we also include 31 truly vulnerable contracts from SmartBugs [11] in this experiment to assess AutoAR's recognition precision.



(a) Precision, Recall, FNR, and FPR by Varying Detection Thresholds (b) Anti-reentrancy Recognition Accuracy w/ 2.3σ Threshold

Fig. 8: AutoAR's Anti-reentrancy Recognition Performance

Recall that AutoAR's recognition model operates with a detection threshold τ, which is defined as a deviation exceeding the mean distance from each clustering centroid. To comprehensively assess AutoAR's anti-reentrancy recognition performance, we evaluate its precision, recall, false positive rates (FPR), and false negative rates (FNR) across different detection thresholds. As shown in Figure 8(a), positive thresholds enable AutoAR to significantly recognize anti-reentrancy-protected contracts with a low FPR. Particularly, the thresholds between $[1.8\sigma, 2.8\sigma]$—where σ represents the standard distance deviation within each cluster—yield recall rates over 85% with 0% FPR. Therefore, selecting thresholds within this range guarantees that AutoAR effectively identifies anti-reentrancy-protected contracts, without the risk of misidentifying vulnerable contracts as *anti-reentrancy-protected*. For subsequent experiments, we conservatively choose 2.3σ, the midpoint of this range, as our detection threshold, which achieves a recall rate of 88% and 100% precision.

Using a detection threshold of 2.3σ, we further assess AutoAR's detailed recognition accuracy for each of the 12 anti-reentrancy patterns, For each pattern, we randomly select 30 contracts marked by AutoAR as likely containing this pattern, resulting in a total of 360 contracts for manual verification. AutoAR is deemed capable of recognizing a specific anti-reentrancy pattern if the related external calls in these contracts are indeed safeguarded by anti-reentrancy patterns. The experimental results, displayed in Figure 8(b), reveal that AutoAR can identify any pattern with an accuracy exceeding 84%, achieving an average accuracy of approximately 89%.

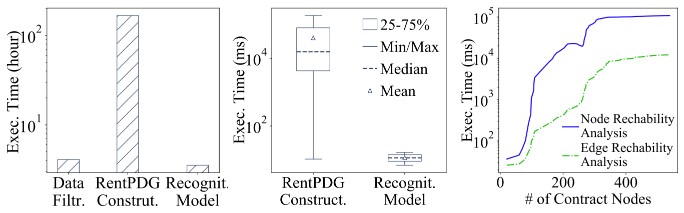
To further demonstrate how effectively AutoAR can enhance the capabilities of existing detectors in reentrancy detection, we have integrated AutoAR into the detection workflows of six tools, as detailed in§VI-B. Using this detection suite, we scan 31 truly vulnerable contracts from SmartBugs [11] and 298 contracts previously confirmed to contain anti-reentrancy patterns. As depicted in Table II, the integration of AutoAR leads to a significant reduction in false positive rates (FPR) for vulnerability detection—by more than 85%—compared to the original detection. Additionally, the fact that 0% reduction in true positives (TPs) across all six detectors confirms that the integration of AutoAR does not compromise the detectors' original ability to identify truly vulnerable contracts.

TABLE II: Integrating AutoAR with 6 Tools to Scan 31 Vulnerable and 298 Non-Vulnerable Contracts

Detectors		Recall	Precision	#TPs	#FPs	FNR	FPR
Slither	Original	1	0.128	31	211	0	0.708
	w/ AutoAR	1	0.596	31	21	0	0.070 ↓**(90%)**
Securify	Original	0.935	0.184	29	129	0.065	0.433
	w/ AutoAR	0.935	0.644	29	16	0.065	0.054 ↓**(88%)**
Mythril	Original	0.323	0.161	10	52	0.677	0.174
	w/ AutoAR	0.323	0.588	10	7	0.677	0.023 ↓**(87%)**
Conkas	Original	1	0.164	31	158	0	0.530
	w/ AutoAR	1	0.564	31	24	0	0.081 ↓**(85%)**
Smartian	Original	0.419	0.283	13	33	0.581	0.111
	w/ AutoAR	0.419	0.867	13	2	0.581	0.007 ↓**(94%)**
Sailfish	Original	0.903	0.184	28	124	0.097	0.416
	w/ AutoAR	0.903	0.636	28	16	0.097	0.054 ↓**(87%)**

E. RQ4: Computational Overhead

To enhance the efficiency of AutoAR, we have implemented data filtration and RentPDG construction using 10 parallel processes, while also utilizing GPU acceleration for training our recognition model. Optimization strategies have been employed to expedite RentPDG construction, as detailed in Appendix B. The overall overhead of AutoAR is depicted in Figure 9. During the training phase, AutoAR filters out 34,081 reentrancy-susceptible contracts from 92K contracts in 4.1 hours, constructs 111,681 RentPDGs in 166.7 hours, and trains a recognition model in 3.2 hours. In the recognition phase, AutoAR takes around 30s on average to construct RentPDGs from a contract and 10ms to use the model for detecting if a contract is protected by anti-reentrancy patterns.



(a) Training Phase for 92k Contracts (b) Recognition Phase Per Contract (c) Detailed Time of RentPDG Construction

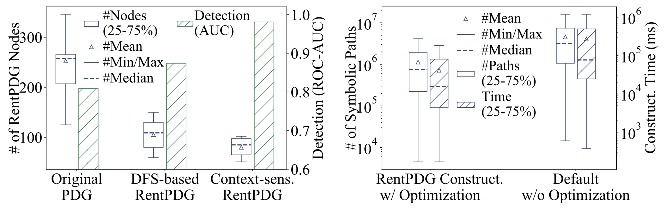
Fig. 9: AutoAR's Computational Overhead

Furthermore, we analyze the detailed overhead of RentPDG construction in the recognition phase. As shown in Figure 9(c), the node and edge reachability analysis time for RentPDG construction increases significantly with the node number. Statistics from real-world smart contracts (Appendix D) indicate that 80% of contracts contain fewer than 300 nodes.

Consequently, in most cases, the time to construct RentPDGs is typically $< 60s$ for node reachability analysis and $< 1s$ for edge reachability analysis, which is acceptable in practice.

F. RQ5: Ablation Studies

We conduct ablation studies to address the impact of our RentPDG presentation and the effectiveness of core components within the RentPDG construction module, namely the context-sensitive reachability analysis (§V-C) and optimization approach (Appendix B). As shown in Figure 10(a), compared to the original PDG representation, RentPDG significantly reduces irrelevant graph nodes and enhances model performance. Moreover, when compared to the naive DFS-based method for constructing RentPDGs, employing context-sensitive reachability analysis effectively eliminates irrelevant nodes by an average of 24% and improves the detection ROC-AUC by 12%. The ROC-AUC is a common metric used to comprehensively assess overall detection performance across different detection thresholds [18]. Additionally, as shown in Figure 10(b), the optimized construction method substantially outperforms the default method, reducing redundant symbolic paths and accelerating construction time by average ratios of 73% and 80%, respectively.



(a) Effect of RentPDG and its Two Construction Methods (b) Effect of Optimized Construction Approach

Fig. 10: Ablation Study for Different RentPDG Components

VIII. Related Work

Reentrancy Detection. The reentrancy vulnerability [52] poses a significant threat to Ethereum smart contracts, potentially leading to substantial financial losses by allowing attackers to manipulate contract states through re-entrant calls. To address this, a range of automated detectors have been developed. Static analysis tools such as Slither [12], Smartcheck [46], and Securify [47] identify reentrancy patterns in Solidity code, SlithIR, EVM-IR, and EVM bytecode, respectively. On the other hand, the dynamic analysis tools such as Conkas [48], Oyente [25], Manticore [30], and Mythril [9] utilize symbolic execution engines to trace contract execution paths and identify reentrancy vulnerabilities. However, empirical studies [11], [36] have shown that these state-of-the-art detectors often report a large number of false positives (FPs), many of which are contracts actually safeguarded by anti-reentrancy patterns.

Anti-reentrancy Patterns. In Solidity contract development, anti-reentrancy patterns have emerged as countermeasures against reentrancy attacks. These anti-reentrancy patterns often constitute proactive coding practices designed to thwart such attacks before they occur. Recent studies [11], [51] indicate that most FPs reported by current reentrancy detectors stem from an oversight of these anti-reentrancy patterns. Thus, recognizing a broad spectrum of proactive anti-reentrancy patterns is crucial for enhancing the capabilities of reentrancy detectors. This includes not only code patterns explicitly designed to counter reentrancy attacks but also those not specifically intended for this purpose, yet still capable of mitigating reentrancy risks.

TABLE III: Literature Review of Anti-reentrancy Patterns

Anti-reentrancy Type	Literature			Total
	Research	Blog	Document	
Checks-Effects-Interactions (CEI)	[29], [27], [54], [51]	[35], [42], [19], [31]	[43]	9
Safe Ether Transfer (P1)	[5], [51]	[19], [42]	-	4
Mutex Variable (P2)	[5], [51]	[19]	-	3
Sender Check (P3)	[51]	-	-	1
Reentrancy Guard (P6)	[51]	[42], [31]	[33]	4
P4-5, P7-12	-	-	-	0

We conduct an extensive literature review of recent articles that discuss anti-reentrancy patterns from various sources: (i) academic research discussing anti-reentrancy patterns; (ii) blog posts analyzing contract security against reentrancy attacks; and (iii) official documentation. This review confirms that only the Checks-Effects-Interactions (CEI), along with P1-3, and P6, are acknowledged as effective against reentrancy attacks. Other newly identified patterns, P4-5 and P7-12, identified in this work are rarely mentioned, as summarized in Table III. Additionally, our investigation reveals a notable absence of automated tools that could assist state-of-the-art detectors in effectively identifying these anti-reentrancy patterns.

Vulnerability Detection via Graph Learning. Rather than treating code as textual information, some vulnerability detection work [24], [50], [56] uses intermediate program graphs to model complex semantic relationships in traditional programs or smart contracts, such as the control flow graph (CFG) and program dependency graph (PDG). However, these graphs may inaccurately capture anti-reentrancy semantics, often including irrelevant information. In response, our RentPDG is tailored to specifically capture the crucial anti-reentrancy-related semantics, particularly control and data dependency around external calls. Additionally, as the graph-based vulnerability detection work typically relies on supervised learning, it is inherently unsuitable for our task, *i.e.,* learning anti-reentrancy patterns on Ethereum without the guidance of ground truth labels.

IX. Discussion

Emergence of New Anti-Reentrancy Patterns. Our AutoAR is trained on existing contracts protected by anti-reentrancy patterns. While this setup enables AutoAR to effectively identify prevalent anti-reentrancy patterns, it may falsely recognize new, unseen anti-reentrancy patterns. This issue is similar to the concept drift problem in anomaly detection. A viable solution is to selectively retrain AutoAR with these false recognition samples, enabling it to adapt quickly to emerging anti-reentrancy patterns in the evolving smart contract landscape.

Applicability. Although AutoAR operates solely on source code, it is highly applicable as recent research [1] indicates that about one-third of contracts are available with source code. For developers and security auditors who often have access to the source code, it provides substantial benefits as it can effectively reduce their false-positive vulnerability reports.

Rule-based Anti-reentrancy Recognition. While static and dynamic analysis tools can define rules to identify known anti-reentrancy patterns, they require prior expertise and offer limited flexibility in detecting unknown patterns. Additionally, our literature review (Table III) reveals a significant gap in the existing knowledge on identifying anti-reentrancy patterns.

Security of Identified Anti-reentrancy Patterns. The P1-3 and P6 patterns are widely recognized as effective measures to prevent reentrancy [5], [51]. Our work identifies eight additional patterns (§VII-B) that achieve similar security outcomes by imposing control/data dependency restrictions. Specifically, P4 and P10 impose the strictest restrictions by prohibiting the passage of untrusted parameters (P4) from untrusted addresses (P10), effectively nullifying the precondition for reentrancy. Meanwhile, P5 and P7-9 allow unauthorized function access but impose constraints on contract code execution (P5), access frequency (P7), financial access cost (P8), and recursive calls (P9), which effectively prevents external contracts from making repeated function calls. Notably, P7 seems ineffective against repeated function calls over long time intervals when a small access frequency constraint is set; however, contract execution timestamps *block.timestamp* elapse at block-level granularity, ensuring that even a small constraint can prevent repeated function access within a block time. Additionally, P11-12 allow repeated function calls but maintain data consistency, offering flexibility while ensuring security.

X. Conclusion

This paper presents AutoAR, an automated recognition system designed to identify prevalent anti-reentrancy patterns in Ethereum smart contracts. By integrating with existing reentrancy detectors, AutoAR can significantly reduce reported false positive cases. To effectively learn anti-reentrancy patterns, AutoAR first selects contracts likely enforced with anti-reentrancy patterns, and then constructs specific graph representations RentPDGs from contracts to capture anti-reentrancy-related code information. Based on the RentPDGs, AutoAR employs an anti-reentrancy recognition model that integrates graph auto-encoder with clustering to precisely learn and identify the key logic of anti-reentrancy patterns. Our experimental results show the efficacy of AutoAR in anti-reentrancy identification.

Acknowledgment

We are grateful to all the anonymous reviewers for their insightful comments, which highly improve our paper. The research is supported in part by the National Key Research and Development Program of China (No.2021YFB2910109) and the National Natural Science Foundation of China (No. 62202465 and 62202260).

References

[1] T. Abdelaziz and A. Hobor, "Smart learning to find dumb contracts," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1775–1792.

[2] R. Behnke. Explained: The fei protocol hack (april 2022). [Online]. Available: https://www.halborn.com/blog/post/explained-the-fei-protocol-hack-april-2022

[3] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2329–2344.

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

[5] 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.

[6] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, no. 37, pp. 2–1, 2014.

[7] J. Chen, W. Han, M. Yin, H. Zeng, C. Song, B. Lee, H. Yin, and I. Shin, "{SYMSAN}: Time and space efficient concolic execution via dynamic data-flow analysis," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2531–2548.

[8] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 227–239.

[9] Consensys, "Mythril," 2023. [Online]. Available: https://github.com/Consensys/mythril

[10] T. H. Cormen, C. E. Leiserson, and Z. Rivest, "Introduction to algorithms," *Resonance*, 1990.

[11] 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 ACM/IEEE 42nd International conference on software engineering*, 2020, pp. 530–541.

[12] J. Feist, G. Grieco, and A. 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)*. IEEE, 2019, pp. 8–15.

[13] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[14] M. Gagolewski, "genieclust: Fast and robust hierarchical clustering," *SoftwareX*, vol. 15, p. 100722, 2021.

[15] Gensim, "Topic modeling for humans," 2023. [Online]. Available: https://radimrehurek.com/gensim/

[16] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 415–427.

[17] X. Han, T. Pasquier, A. Bates, J. Mickens, and M. Seltzer, "Unicorn: Runtime provenance-based detector for advanced persistent threats," in *Network and Distributed System Security Symposium*, 2020.

[18] J. Huang and C. X. Ling, "Using auc and accuracy in evaluating learning algorithms," *IEEE Transactions on knowledge and Data Engineering*, vol. 17, no. 3, pp. 299–310, 2005.

[19] insurgent, "Solidity Smart Contract Security: 4 Ways to Prevent Reentrancy Attacks," May 2022. [Online]. Available: https://betterprogramming.pub/solidity-smart-contract-security-preventing-reentrancy-attacks-fc729339a3ff

[20] J. Kanani, S. Nailwal, and A. Arjun, "Matic whitepaper," *Polygon, Bengaluru, India, Tech. Rep., Sep*, 2021.

[21] J. H. Lau and T. Baldwin, "An empirical evaluation of doc2vec with practical insights into document embedding generation," in *Proceedings of the 1st Workshop on Representation Learning for NLP, Rep4NLP@ACL 2016, Berlin, Germany, August 11, 2016*, P. Blunsom, K. Cho, S. B. Cohen, E. Grefenstette, K. M. Hermann, L. Rimell, J. Weston, and S. W. Yih, Eds. Association for Computational Linguistics, 2016, pp. 78–86. [Online]. Available: https://doi.org/10.18653/v1/W16-1609

[22] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[23] F. Liu, Y. Wen, D. Zhang, X. Jiang, X. Xing, and D. Meng, "Log2vec: A heterogeneous graph embedding based approach for detecting cyber

threats within enterprise," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 1777–1794.

[24] F. Luo, R. Luo, T. Chen, A. Qiao, Z. He, S. Song, Y. Jiang, and S. Li, "Scvhunter: Smart contract vulnerability detection based on heterogeneous graph attention network," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[25] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[26] E. Manzoor, S. M. Milajerdi, and L. Akoglu, "Fast memory-efficient anomaly detection in streaming heterogeneous graphs," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 1035–1044.

[27] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "Verisolid: Correct-by-design smart contracts for ethereum," in *Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23*. Springer, 2019, pp. 446–465.

[28] M. I. Mehar, C. L. Shier, A. Giambattista, E. Gong, G. Fletcher, R. Sanayhie, H. M. Kim, and M. Laskowski, "Understanding a revolutionary and flawed grand experiment in blockchain: the dao attack," *Journal of Cases on Information Technology (JCIT)*, vol. 21, no. 1, pp. 19–32, 2019.

[29] A. Mense and M. Flatscher, "Security vulnerabilities in ethereum smart contracts," in *Proceedings of the 20th international conference on information integration and web-based applications & services*, 2018, pp. 375–380.

[30] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.

[31] MythX, "SWC-107 - Smart Contract Weakness Classification (SWC)." [Online]. Available: https://swcregistry.io/docs/SWC-107/

[32] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized business review*, 2008.

[33] OpenZeppelin, "Security - OpenZeppelin Docs." [Online]. Available: https://docs.openzeppelin.com/contracts/4.x/api/security

[34] Openzeppelin, "The standard for secure blockchain applications," 2023. [Online]. Available: https://www.openzeppelin.com/

[35] OWASP, "Reentrancy Attacks." [Online]. Available: https://owasp.org/www-project-smart-contract-top-10/2023/en/src/SC01-reentrancy-attacks.html

[36] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh, "Empirical vulnerability analysis of automated smart contracts security testing on blockchains," *arXiv preprint arXiv:1809.02702*, 2018.

[37] pcaversaccio, "A historical collection of reentrancy attacks," 2023. [Online]. Available: https://github.com/pcaversaccio/reentrancy-attacks

[38] D. Perez and B. Livshits, "Smart contract vulnerabilities: Vulnerable does not imply exploited," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1325–1341.

[39] S. K. Prasad, S. K. Das, and C.-Y. Chen, "Efficient erew pram algorithms for parentheses-matching," *IEEE transactions on parallel and distributed systems*, vol. 5, no. 9, pp. 995–1008, 1994.

[40] Z. Pratap, "Reentrancy Attacks and The DAO Hack Explained | Chainlink." [Online]. Available: https://blog.chain.link/reentrancy-attacks-and-the-dao-hack/

[41] T. Reps, S. Schwoon, and S. Jha, "Weighted pushdown systems and their application to interprocedural dataflow analysis," in *International Static Analysis Symposium*. Springer, 2003, pp. 189–213.

[42] R. |. Secureum, "Smart Contract Security 101 — Secureum #7," Feb. 2021. [Online]. Available: https://secureum.substack.com/p/smart-contract-security-101-secureum

[43] Solidity, "Solidity documentation," 2023. [Online]. Available: https://docs.soliditylang.org/en/v0.8.21/

[44] Y. Sui, X. Cheng, G. Zhang, and H. Wang, "Flow2vec: Value-flow-based precise code embedding," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.

[45] F. Tian, B. Gao, Q. Cui, E. Chen, and T.-Y. Liu, "Learning deep representations for graph clustering," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 28, no. 1, 2014.

[46] 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.

[47] 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.

[48] N. Veloso and I. S. Técnico, "Conkas: a modular and static analysis tool for ethereum bytecode," 2021.

[49] C. Wang, S. Pan, G. Long, X. Zhu, and J. Jiang, "Mgae: Marginalized graph autoencoder for graph clustering," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, 2017, pp. 889–898.

[50] H. Wu, Z. Zhang, S. Wang, Y. Lei, B. Lin, Y. Qin, H. Zhang, and X. Mao, "Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 378–389.

[51] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, and T. Peng, "Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1029–1040.

[52] M. Zhang, X. Zhang, Y. Zhang, and Z. Lin, "{TXSPECTOR}: Uncovering attacks in ethereum from transactions," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2775–2792.

[53] X. Zhang, X. Wang, R. Slavin, and J. Niu, "Condysta: Context-aware dynamic supplement to static taint analysis," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 796–812.

[54] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, and D. Gu, "Smartshield: Automatic smart contract protection made easy," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 23–34.

[55] Z. Zheng, N. Zhang, J. Su, Z. Zhong, M. Ye, and J. Chen, "Turn the rudder: A beacon of reentrancy detection for smart contracts on ethereum," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 295–306.

[56] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

Appendix A
Data Filtration Algorithms

A. Recognizing Reentrancy-susceptible Contracts

To identify contracts susceptible to cross-function and cross-contract reentrancy attacks, our analysis encompasses not only the inter-procedural contexts of internal functions but also typical API calls to external functions. These API functions, residing in well-known contracts, offer universally accessible services such as exchanging ERC20 tokens for ETH. In our approach, we focus exclusively on those functions that interact with and have callbacks to invoking contracts, as these interactions can affect the states of the invoking contracts. As shown in Table IV, we analyze 22 commonly used API functions, categorized into those related to ERC20 and ERC721 tokens. For these functions, we disregard their internal states and focus solely on their interactions with the invoking contracts, specifically the callbacks.

Optimization Approach. A straightforward method to identify reentrancy-susceptible contracts involves examining each inter-procedural path. However, this process can be time-consuming

TABLE IV: Considered API Function Calls

Type	API Functions	Total
ERC20	IUniswapV2Router02.{addLiquidity, addLiquidityETH, removeLiquidity, removeLiquidityETH, removeLiquidityWithPermit, removeLiquidityETHWithPermit, swapExactTokensForTokens, swapTokensForExactTokens, swapExactETHForTokens, swapTokensForExactETH, swapExactTokensForETH, swapETHForExactTokens, removeLiquidityETHSupportingFeeOnTransferTokens, removeLiquidityETHWithPermitSupportingFeeOnTransferTokens, swapExactTokensForTokensSupportingFeeOnTransferTokens, swapExactETHForTokensSupportingFeeOnTransferTokens, swapExactTokensForETHSupportingFeeOnTransferTokens}	15
ERC721	Seaport.{fulfillBasicOrder, fulfillOrder, fulfillAdvancedOrder, fulfillAvailableOrders, fulfillAvailableAdvancedOrders, fulfillAvailableOrders, cancel}	7

Algorithm 1: Recognize Reentrancy-Susceptible Contract

```
Input: a smart contract $ct$.
Output: a boolean variable.
1 foreach $f \in ct$ do
2     DfsContext($f.entry$, $ct$); // see Algorithm 2
3 foreach call $c \in ct$ do
4     if $c.readBefore \cap (c.writeBefore \cup c.writeAfter)$ then
5         return True;
6 return False;
```

due to the potentially vast number of inter-procedural paths. To streamline this, we introduce an optimization approach that involves truncating inter-procedural paths and merging certain sub-paths. Specifically, this method divides each inter-procedural path into multiple intra-procedural paths. For a function f, it merges some paths outside the function and disregards irrelevant paths, *e.g.,* paths not containing external calls or state write/read operations. By integrating these merged paths with the intra-function paths of f, we can efficiently assess whether the statements in these paths satisfy Pattern 1.

Algorithm 1 illustrates this optimization approach. Initially, we enumerate all functions and modifiers in a contract ct and examine their intra-procedural paths. Using a recursive method, *DfsContext* (Algorithm 2), we walk through each intra-procedural path to analyze pertinent context information, such as state write and read operations within ct. For each statement t, we consolidate context information from relevant paths, including those preceding in caller functions and the intra-procedural paths in f. Specifically, the context information covers state read and written statements triggered by t within the current contract ct. Subsequently, we perform static analysis on call statements and determine if the corresponding context information satisfies Pattern 1. As depicted in Line 4, for an external call statement c, if state read operations occur in its pre-context, and state write operations in either its pre-context or post-context, we output *True*, indicating the contract's susceptibility to reentrancy attacks. Conversely, if no call statements satisfy this condition, we output *False*.

Algorithm 2: DfsContext: Context Analysis

```
Input: a statement $t$ and a related contract $ct$.
Output: context information of $t$.
1 if $t$ is visited then
2     return;
3 $V.add(t)$, fRead = $\emptyset$, sWrite = $\emptyset$;
4 foreach $p \in t.fathers$ do // merge pre-context info
5     fRead $\cup$ = $p$.readBefore;
6 if fRead $\not\subset$ $t.readBefore$ then //bring new context
7     $t$.readBefore $\cup$ = fRead $\cup$ $t$.allStateReadIn(ct);
8     foreach invoked internal/API function $f \in t$ do
9         foreach $s_f \in f.allStatementsIn(ct)$ do
              // merge caller context with the
                 current f's statements
10            $s_f$.readBefore $\cup$ = fRead
11 foreach $sn \in t.sons$ do // merge post-context
12     DfsContext(sn);
13     $t$.writeAfter$\cup$=$s$.writeAfter $\cup s$.allStateWriteIn(ct)
```

Algorithm 3: Recognize Contract in Final Dataset

```
Input: a smart contract $ct$.
Output: a set of external calls $S_{call}$ that are
        susceptible to reentrancy attacks.
1 foreach $f \in ct.modifierAndFuncs$ do
2     DfsContext($f.entry$); // see Algorithm 2
3 foreach call $c \in ct$ do
4     if $c.readBefore \cap c.writeAfter$ then
5         $S_{call}$.add($c$);
6 return $S_{call}$;
```

B. Recognizing Final Dataset

According to Pattern 3, we can easily recognize target contracts via static analysis. Algorithm 3 shows this recognition process. This algorithm takes a contract ct as input and then outputs a set S_{call} of reentrancy-susceptible external calls. If S_{call} is non-empty, we then include the corresponding contract in our final dataset. Notably, the set of reentrancy-susceptible calls S_{call} can be used in subsequent RentPDG construction since this process needs to analyze nodes and edges that reach the calls.

In this algorithm, we first enumerate functions and modifiers in a contract and use *DfsContext* (Algorithm 2) to analyze inter-procedural context information. Next, we examine external call statements' context information via static analysis. For each external call statement c, if read and write operations for a state variable exist in its pre-context and post-context respectively, we can deem that c is susceptible to reentrancy attacks and its context information satisfies Pattern 3. Then, we add the external call statement into a set S_{call}. When all external call statements' context information is examined, this algorithm finally outputs S_{call}.

Algorithm 4: Path Compression

```
Input: a symbolic path $p$.
Output: a compressed path $p'$.
1  foreach $c \in p$ do
2      if c.label == '(' then stack.push(c) ;
3      if c.label == 'e' and stack.top != 'M' then
4          stack.push('M')
5      if c.label == ')' then
6          while stack.top.label != '(' do stack.pop();
7          if not Matched(stack.top, c) then return $\emptyset$ ;
8          stack.pop();
9          stack.push('M'); // $(_i$ matches $)_i$
10 return stack.symbols;
```

Appendix B
Optimized RentPDG Construction

To construct RentPDGs, we need to perform a series of symbolic matrix-vector multiplication and element-wise product operations to compute node reachability vectors and an edge reachability matrix. Notably, all of these matrix operations are composed of symbolic multiplication/addition operations, each of which executes in an amount of time linear to the number of symbolic paths (see Equation 8). Therefore, the execution time of our context-sensitive reachability analysis is determined by the number of involved symbolic paths. In practice, a contract program dependence graph (PDG) always contains a large number of inter-procedural paths, so it may be time-consuming to perform context-sensitive reachability analysis. To address this, we provide the following optimization approaches to reduce symbolic paths in matrix operations.

A. Path Compression

For a symbolic node reachability vector, the i-th element is a set of symbolic paths from node i to c. Given that some symbolic paths may have similar grammar contexts, we can compress them into one short expression and speed up the corresponding symbolic matrix operations. For instance, the symbolic paths $\underline{e(_1e)_1e})_3$ and $\underline{e(_2e)_2e})_3$ have similar parenthesis-balanced substrings in the front. According to the CFL grammar, we can compress the symbolic paths to the same symbolic expression $B)_3$, where B represents their parenthesis-balanced substrings, *i.e.,* $e(_1e)_1e$ and $e(_2e)_2e$. Based on this observation, we can replace all parenthesis-balanced substrings with B to compress symbolic paths.

Inspired by parenthesis checker algorithms, we can leverage a stack to identify parenthesis-balanced parts of a symbolic path and then replace them with the symbol B. According to the CFL grammar, the intra-procedural parts of a symbolic path are also considered parenthesis-balanced, *e.g.,* ee and e. They can be also represented by the symbol B. Algorithm 4 shows the process of path compression. The key idea is to scan a symbolic path from left to right and put all opening parentheses into the stack.

When we hit the symbol e, we check the stack top. If it is not M, we push M into the stack. When we hit a closing parenthesis, we pop up all symbols until the opening parenthesis is encountered. If the opening parenthesis matches the closing parenthesis, we then pop up it and push M into the stack. Otherwise, the symbolic path is considered infeasible. Finally, the remaining symbols in the stacks are output as the compressed path.

B. Path Pruning

Definition 3 indicates that we do not need to check all paths between node pairs to recognize node reachability information. If we find one feasible path from node i to c, we say that n_i can reach c in the calling context. To reduce the reachability analysis time, we can preserve the most likely feasible paths and remove other paths from node reachability vectors. For a symbolic node reachability vector $\overline{\boldsymbol{\mu}}^{\boldsymbol{c}}$, it may contain two types of symbolic paths of length l in the i-th element: (i) parenthesis-balanced paths B and (ii) partially parenthesis-balanced paths U. After performing one-step matrix-vector multiplication $\mathbf{A_s} \cdot \overline{\boldsymbol{\mu}}^{\boldsymbol{c}}$, the result vector may contain two types of symbolic representation, either xB or xU, of $(l+1)$ length. Here, x refers to a symbolic variable, which can be instantiated by three symbols: $(_i$, $)_i$, and e.

Compared to xU, the symbolic path type xB is more likely to be feasible. This is because all possible instances for xB, for instance, $(_iB$, $)_iB$, and eB, satisfy the CFL grammar. On the other hand, one possible instance for xU, *e.g.,* $(_iU$, may not satisfy the grammar. Therefore, we can only preserve xB and exclude xU from the reachability vector $\overline{\boldsymbol{\mu}}^{\boldsymbol{c}}_{\boldsymbol{h}}$ in the node reachability analysis. According to this observation, we can prune all partially parenthesis-balanced paths U from node reachability vectors to speed up our reachability analysis.

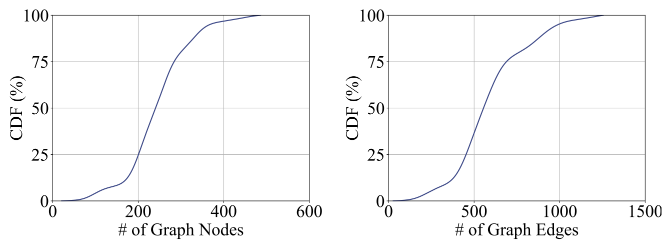
Appendix C
Node Semantic Attributes

We show the 60 semantic attributes for a node in Table V. Overall, the features can be categorized into six main types.

- *Variables and Literals (17).* This type includes an attribute related to constants, 4 attributes related to variable numbers, 8 attributes related to variable types, and 4 attributes indicating if there are special solidity variables.
- *Function Calls (6).* This type covers an attribute related to modifier calls, 2 attributes related to internal function calls, 2 attributes related to external function calls, and an attribute indicating if there is a function with Ether transfer.
- *Built-in API Calls (12).* We use an attribute to describe the number of built-in Ether transfer functions and use 11 attributes to reveal the number of built-in functions related to access control.
- *Node Type (18).* We use 18 attributes to describe the type of a node in control flows. Each attribute indicates whether the node belongs to a node type.
- *Library API Calls (3).* We use an attribute to reveal the number of library API calls and two attributes to indicate if there exist verification and context function calls.
- *Operators (4).* This type includes the attributes revealing relation, logic, arithmetic, and bitwise operators.

TABLE V: Node Semantic Attributes

Types	No.	Attributes
Variables & Litereals	100	revealing # of constant numbers, including constant literals and identifiers.
	101-104	revealing # of local variables read and written, and global variables read and written.
	105-112	revealing # of integer, array, boolean, bytes, contract, address, solidity built-in, and private variables.
	112-116	indicating the existence of tx.origin, msg,sender, block.timestamp, and msg.value.
Operators	117-120	revealing # of relation, logic, arithmetic, and bitwise operators.
Function Calls	121	revealing # of function modifier calls.
	122-123	revealing # of public and private internal function calls.
	124-125	revealing # of high-level and low-level external function calls.
	126	indicating the existence of an external call function that sends Ether.
Built-in API Calls	127	revealing # of built-in Ether transfer functions, including *send()* and *transfer()*
	128-138	revealing # of invoked *require(), revert(), keccack256(), ecrecover(), selfdestruct(), suicide(), ripemd160(), blockhash(), balance(), abi.encode(), and abi.encodePacked().*
Node Type	139-156	indicating the node types: entry point, expression, return, condition, variable declaration, assembly, loop condition, end-if, loop start, loop end, throw, break, continue, placeholder, try, catch, state entry point, and contract start.
Library API Calls	157	revealing # of library function calls.
	158	indicating the existence of verification functions, *i.e., MerkleProof.verify()* and *ECDSA.recover().*
	159	indicating the existence of some context functions, *i.e., owner()* and *_msgSender().*



(a) CDF of Graph Nodes (b) CDF of Graph Edges

Fig. 11: CDFs of Nodes and Edges in Ethereum Contract PDGs

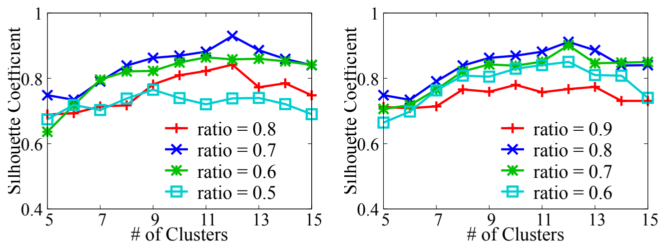
Appendix D
Statistics Properties of Ethereum Contracts

Here, we delve deeper into the complexity of our contract datasets. Given that the performance of our system—particularly the component of RPDG construction—is significantly influenced by cross-function control and data dependencies in contracts, we analyze the statistical properties of inter-procedural program dependency graphs (PDGs) of these contracts. We evaluate the cumulative distribution functions (CDFs) of nodes and edges. As demonstrated in Figure 11, 80% of real-world contracts contain fewer than 300 nodes and 800 edges.

Appendix E
Determining System Parameters

Several parameters critically influence our system's performance, particularly within the graph auto-encoder and the clustering algorithm. The down-sampling and pooling ratios significantly impact the quality of graph-level embeddings, while the number of clusters affects the quality of clustering results. To identify optimal settings, we employ the silhouette coefficient method, which is widely used in unsupervised learning tasks [17], [26], [23]. This method facilitates testing various parameter configurations to achieve the best clustering quality. Typically, a higher average silhouette coefficient indicates better clustering quality. To find the most effective configurations, we experiment with varying numbers of clusters, down-sampling, and pooling ratios, monitoring their impact on the average silhouette coefficients.

Down-sampling Ratio. In our graph auto-encoder, the down-sampling ratio, denoted as r_d, indicates the reduction in node feature dimensions at each convolutional layer. This ratio determines the final graph-level embedding dimension ($320r_d^3$). Figure 12(a) shows how the average silhouette coefficient varies across different down-sampling ratios. The coefficient peaks at a ratio of 0.7, suggesting that embeddings with a dimensionality of 110 yield the best clustering outcomes. Lower ratios, such as 0.5, appear to compromise performance, likely due to the loss of significant semantic information.



(a) Down-sampling Ratio (b) Pooling Ratio

Fig. 12: Silhouette Coefficients with Different Parameters

Pooling Ratio. The pooling ratio specifies the proportion of nodes retained at each pooling layer, which directly influences the number of node-level embeddings produced by the encoder. As detailed in Equation 13, an increased number of node-level embeddings implies that the final graph-level embedding will contain more semantic information. Figure 12(b) illustrates the trend of average silhouette coefficients for four different pooling ratios. We observe that a pooling ratio of 0.8 yields the highest average silhouette coefficients. In contrast, a ratio of 0.9, despite retaining a greater amount of semantic information, leads to lower coefficients. This decline may be due to an excess of semantic information at this ratio, potentially adversely affecting the final clustering quality.

Number of Clusters. As evident from Figure 12, 12 clusters achieve the highest average silhouette coefficient with a down-sampling ratio of 0.7 and a pooling ratio of 0.8.