

Copy-and-Paste? Identifying EVM-Inequivalent Code Smells in Multi-chain Reuse Contracts

ZEXU WANG, Sun Yat-sen University, China and Peng Cheng Laboratory, China

JIACHI CHEN, Sun Yat-sen University, China and Guangdong Engineering Technology Research Center of Blockchain, China

TAO ZHANG, Macau University of Science and Technology, China

YU ZHANG, Harbin Institute of Technology, China and Peng Cheng Laboratory, China

WEIZHE ZHANG, Harbin Institute of Technology, China and Peng Cheng Laboratory, China

YUMING FENG, Peng Cheng Laboratory, China

ZIBIN ZHENG*, Sun Yat-sen University, China and Guangdong Engineering Technology Research Center of Blockchain, China

As the development of *Solidity* contracts on *Ethereum*, more developers are reusing them on other compatible blockchains. However, developers may overlook the differences between the designs of the blockchain system, such as the *Gas Mechanism* and *Consensus Protocol*, leading to the same contracts on different blockchains not being able to achieve consistent execution as on *Ethereum*. This inconsistency reveals design flaws in reused contracts, exposing code smells that hinder code reusability, and we define this inconsistency as *EVM-Inequivalent Code Smells*.

In this paper, we conducted the first empirical study to reveal the causes and characteristics of *EVM-Inequivalent Code Smells*. To ensure the identified smells reflect real developer concerns, we collected and analyzed 1,379 security audit reports and 326 *Stack Overflow* posts related to reused contracts on EVM-compatible blockchains, such as *Binance Smart Chain* (BSC) and *Polygon*. Using the *open card sorting* method, we defined six types of *EVM-Inequivalent Code Smells*. For automated detection, we developed a tool named *EquivGuard*. It employs static taint analysis to identify key paths from different patterns and uses symbolic execution to verify path reachability. Our analysis of 905,948 contracts across six major blockchains shows that *EVM-Inequivalent Code Smells* are widespread, with an average prevalence of 17.70%. While contracts with code smells do not necessarily lead to financial loss and attacks, their high prevalence and significant asset management underscore the potential threats of reusing these smelly *Ethereum* contracts. Thus, developers are advised to abandon *Copy-and-Paste* programming practices and detect *EVM-Inequivalent Code Smells* before reusing *Ethereum* contracts.

CCS Concepts: • **Software and its engineering** → *Software testing and debugging*.

Additional Key Words and Phrases: Code reuse, Smart contract, Code smell, Static taint analysis

*Corresponding Author

Authors' Contact Information: Zexu Wang, Sun Yat-sen University, Zhuhai, China and Peng Cheng Laboratory, Shenzhen, China, wangzx97@mail2.sysu.edu.cn; Jiachi Chen, Sun Yat-sen University, Zhuhai, China and Guangdong Engineering Technology Research Center of Blockchain, Zhuhai, China, chenjch86@mail.sysu.edu.cn; Tao Zhang, Macau University of Science and Technology, Macau, China, tazhang@must.edu.mo; Yu Zhang, Harbin Institute of Technology, Harbin, China and Peng Cheng Laboratory, Shenzhen, China, yuzhang@hit.edu.cn; Weizhe Zhang, Harbin Institute of Technology, Harbin, China and Peng Cheng Laboratory, Shenzhen, China, wzzhang@hit.edu.cn; Yuming Feng, Peng Cheng Laboratory, Shenzhen, China, fengym@pcl.ac.cn; Zibin Zheng, Sun Yat-sen University, Zhuhai, China and Guangdong Engineering Technology Research Center of Blockchain, Zhuhai, China, zhizbin@mail.sysu.edu.cn.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2994-970X/2025/7-ARTISSTA046

<https://doi.org/10.1145/3728921>

ACM Reference Format:

Zexu Wang, Jiachi Chen, Tao Zhang, Yu Zhang, Weizhe Zhang, Yuming Feng, and Zibin Zheng. 2025. Copy-and-Paste? Identifying EVM-Inequivalent Code Smells in Multi-chain Reuse Contracts. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA046 (July 2025), 23 pages. <https://doi.org/10.1145/3728921>

1 Introduction

As one of the most active blockchains, *Ethereum* provides a platform for the development and deployment of contracts using the *Ethereum Virtual Machine* (EVM) [45, 56]. Currently, *Ethereum* has deployed over 65 million contracts and provided a comprehensive ecosystem for developers. To enable better reusability of contracts on *Ethereum*, more than 60% of blockchains support the execution of EVM, such as *Binance Smart Chain* (BSC) [48], *Polygon* [50], and *Arbitrum* [4], which have already captured a significant market share [27, 33].

However, developers may overlook the differences between blockchains when reusing contracts, which can even lead to financial loss. Specifically, many blockchains have unique local running environments and settings that differ from *Ethereum*. These differences may result in the same contract code producing inconsistent executions across various EVM-compatible blockchains. For example, *Chain ID* [51] uniquely identifies a blockchain network, used to distinguish it from others; directly redeploying *Ethereum* contracts without considering these differences can lead to security issues, as seen in the \$20 million OP token loss by *Wintermute's* wallet [26]. The lack of *Chain ID* validation in *Wintermute's* *Ethereum* contract allowed attackers to exploit the reused contract on *Optimism* blockchain through signature replay attacks, eventually resulting in huge financial loss.

Thus, it is important to ensure the reusability and reliability when reusing smart contracts to other blockchains. In software engineering, *code smells* are characteristics in the source code that may indicate deeper issues, representing design flaws in the program [42]. These flaws can impact code reusability, understandability, and reliability. The inconsistent execution of reused contracts, due to design flaws leading to low reusability and reliability, is becoming increasingly problematic.

This work is the first to study the inconsistency problem from reused *Ethereum* contracts, termed as *EVM-Inequivalent Code Smells*. To ensure the identified smells reflect real developer issues and guide secure contract reuse, we conducted an empirical study to uncover the causes and characteristics of *EVM-Inequivalent Code Smells*. Specifically, we analyzed security audit reports and *Stack Overflow* posts to collect security issues related to reused contracts on EVM-compatible blockchains, such as *Binance Smart Chain* (BSC) [48] and *Polygon* [50]. Through extensive data collection, we gathered 1,379 security audit reports and 326 relevant posts. Using the *open card sorting* method [52], we identified six types of *EVM-Inequivalent Code Smells*: *Cross-Chain Replay Attack* (CCRA), *Time Discrepancy Trap* (TDT), *Fixed Gas Reentrancy* (FGR), *Block Height Misalignment* (BHM), *Phishing Contract Address* (PCA), and *Gas Limit Imbalance* (GLI) (details in Section 3.3). These code smells hinder the reusability and introduce critical security risks for multi-chain reuse contracts.

To investigate the prevalence of the defined six *EVM-Inequivalent Code Smells*, we design an automated detection tool named *EquivGuard*, which utilizes static taint analysis to trace key paths of tainted variable propagation, then employs symbolic execution to verify path reachability and reports detection results. To analyze different code smells, *EquivGuard* identifies key paths using *Domain-Specific Patterns* that encapsulate domain knowledge for identification (details in Section 4.5). Specifically, it uses these patterns to pinpoint tainted sources and sinks, analyze taint propagation through global state dependencies, and verify processing safety before reaching sinks. Based on key paths, *EquivGuard* collects path constraints and uses the *SMT* constraint solver to enable path property verification and report detection results.

As *EVM compatibility* has been adopted by many blockchains, we have applied *EquivGuard* to a dataset of 905,948 contracts across six mainstream blockchains, including *Ethereum*, *BSC*, *Arbitrum*,

Polygon, *Optimism*, and *Avalanche*. Our analysis revealed that 17.70% of the contracts contained at least one *EVM-Inequivalent Code Smell*, indicating the widespread presence of such code smells in the real world. To evaluate *EquivGuard*'s detection performance for *EVM-Inequivalent Code Smells*, we randomly sampled 444 positive labels and 96 negative labels from the detection results. Through manual analysis, *EquivGuard* achieved an overall precision of 95.29% and a recall of 95.83%. Although contracts with code smells do not necessarily lead to financial loss and attacks, their high prevalence and significant asset management highlight the potential threats associated with reusing these smelly *Ethereum* contracts.

The main contributions of this work are as follows:

- To the best of our knowledge, this study represents the first investigation of *EVM-Inequivalent Code Smells*. We defined six types of code smells by manually analyzing real-world security audit reports and *Stack Overflow* posts, providing definitions and examples for each smell.
- We designed *EquivGuard*, combining static taint analysis and symbolic execution. *EquivGuard* achieves an overall 95.29% detection precision, offering an effective solution for the detection of *EVM-Inequivalent Code Smells*.
- We provide a dataset of real-world contracts containing *EVM-Inequivalent Code Smells*. By detecting 905,948 contracts, we found that 17.70% of the contract source codes contained at least one such code smell. While not all affected contracts necessarily lead to financial loss or attacks, this research encourages developers to recognize these smells and prioritize comprehensive testing across varied environments before reuse, mitigating risks from EVM-inequivalent execution.

2 Background

2.1 Explanations of Terminologies

■ **EOAs and Contract Accounts.** *Ethereum* has two account types: External Owned Accounts (EOAs), controlled by private keys, and *Contract Accounts*, whose permissions are determined by their code logic [39]. While EOAs use the same private key across EVM-compatible blockchains, *Contract Accounts* require separate permission setup on each blockchain, potentially leading to permission loss if not properly configured.

■ **Address.** Address is an identifier for EOA or *Contract Account* locations on the blockchain [51]. EOA's addresses are generated by applying the elliptic curve algorithm to a public key and then hashing. *Contract Accounts*' addresses are generated by concatenating the creator's address with the creation transaction's nonce and then hashing the result. This means that a contract address may deploy different code on EVM-compatible blockchains. For example, the contract code at address 0x818ec0a7¹ differs between *Ethereum* and *Moonbeam*.

■ **EIPs.** Ethereum Improvement Proposals (EIPs), like *EIP-2612* which introduced the *permit()* function for offline signature validation [30], aim to change or update *Ethereum* by enhancing its capabilities and security features. However, many EVM-compatible blockchains do not fully comply with EIPs, potentially impacting their EVM compatibility and leading to inconsistencies in how certain features and functions are implemented across different platforms. This lack of standardization can pose challenges for developers aiming for cross-chain interoperability and uniformity in smart contract behavior.

■ **Gas Mechanism.** Gas measures the computational effort on the blockchain. Gas cost quantifies the amount of gas required to execute each operation, e.g., *Ethereum*'s *ADD* operation consumes 3 gas, according to the Gas cost standard [22]. This standard continuously evolves, for instance, *EIP-1380* [3] reduced the Gas cost for the call to self from 700 to 40. Additionally, many EVM-compatible

¹0x818ec0a7fe18ff94269904fced6ae3dae6d6dc0b

blockchains have their own Gas cost standards and refund mechanisms to offer lower transaction fees, such as BSC's BEP [8].

■ **Transfer()/Send().** *Transfer()* and *Send()* functions are used for token transfers, with a fixed 2300 gas limit commonly used to prevent Reentrancy attacks. However, this security measure depends on the Gas cost remaining unchanged [15].

■ **Hard Fork.** Hard Fork significantly changes blockchain protocols, resulting in a split that creates a new blockchain [10]. This process is often used to implement major changes, leading to two versions of the blockchain with distinct paths. This split allows the new protocol to integrate advanced features or security improvements, though it requires consensus to avoid fragmentation.

■ **Block Time.** Block time is the average time it takes for a new block to be added to a blockchain [12].

2.2 EVM-compatible Blockchains







EVM-compatible blockchains support EVM and *Solidity* contracts, enabling developers and users to build DApps across multiple blockchains and reducing barriers to contract deployment and interaction [33]. Table 1 presents the statistics of EVM-compatible blockchain assets on *DefiLlama*. Out of the total, 150 are EVM-compatible blockchains, accounting for 63.03%. The *Total Value Locked* (TVL) of these EVM-compatible blockchains reached \$90.997 billion, significantly surpassing the \$14.213 billion of non-EVM-compatible blockchains. These findings show that EVM-compatible blockchains hold over 86% of the market share, underscoring the importance of EVM compatibility for blockchain networks.

Table 1. Distribution of Chains and Total Value Locked (TVL)

	Chains	Percentage	TVL (billion)	TVL Percentage
EVM-compatible	150	63.03%	90.997	86.49%
Non-EVM-compatible	88	36.97%	14.213	13.51%

We compared the fees and speeds of transactions on the top 5 EVM-compatible blockchains by market share with *Ethereum* (average data from September 2024). As shown in Table 2, BSC's *Time To Finality* (TTF) is only 7.5s, and the average transaction fee on *Polygon* is minimal, at \$0.013. While EVM-compatible blockchains offer lower transaction fees or faster speeds, many blockchain projects must restructure their native virtual machine and continuously update improvement proposals to achieve EVM compatibility. Most blockchains only partially follow or do not follow *Ethereum Improvement Proposals* (EIPs), with only *Polygon* fully supporting them. These modifications and inconsistent implementations hinder the full adaptation of the EVM module on these blockchains, affecting EVM compatibility.

Table 2. Statistics of Top 5 EVM-Compatible Blockchains

Blockchain	Txn Fee	Block Time	TTF	EIP-compliant
 <i>Ethereum</i>	\$15.59	12.14s	16m	Fully
 BSC	\$0.24	3.01s	7.5s	Partial
 <i>Polygon</i>	\$0.013	2.26s	4m16s	Fully
 <i>Arbitrum</i>	\$0.28	0.26s	16m	Partial
 <i>Optimism</i>	\$0.069	2s	16m	Partial
 <i>Avalanche</i>	\$0.23	2.04s	0s	Not

*TTF refers to the time it takes for a transaction to be confirmed.

3 EVM-Inequivalent Code Smells

In this section, we conduct an empirical study on real-world security audit reports and *Stack Overflow* posts related to EVM-compatible blockchains to define and classify *EVM-Inequivalent Code Smells* in reuse contracts.

3.1 Data Collection

To comprehensively analyze real-world *EVM-Inequivalent Code Smells*, we collected 1,379 publicly available audit reports from 30 security teams and 326 relevant *Stack Overflow* posts. We then conducted a thorough manual screening to filter the relevant information for further analysis.

3.1.1 Security Audit Reports Collection. The *Security Audit Report* (SAR) contains specific vulnerabilities and cause analyses, providing a comprehensive understanding of security issues. To collect audit reports related to *EVM-Inequivalent Code Smells* in the real world, we accessed the websites of 81 smart contract security teams listed on *Etherscan* [21] and identified 30 teams that had open-sourced their audit reports, including *SlowMist* [35], *ConsenSys* [16], *BlockSec* [7], and *Trail of Bits* [40]. Additionally, based on non-zero *Total Value Locked* (TVL) and using Solidity contracts, we filtered out 150 EVM-related blockchains from the 207 blockchains listed on *DefiLlama*. Combining this with audit report public websites [18], we collected a total of 1,379 EVM-related audit reports.

3.1.2 Stack Overflow Posts Collection. *Stack Overflow* [37] is a popular Q&A community for developers, engineers, and technology enthusiasts. The platform uses a tag system to manage topics involved in the questions, allowing us to collect the issues and concerns encountered by developers quickly. To ensure efficient analysis of *Stack Overflow Posts* (SOP) related to EVM inequivalence, we initially collected 2,124 posts using the tags “*Solidity Contract*”, “*EVM*” to gather posts related to *Solidity contracts* on *EVM chains*. Subsequently, to filter out irrelevant information and balance human efforts, we used the keywords “*EVM Equivalent*” and “*EVM Compatible*” for further screening, resulting in 326 related posts on contract reuse in EVM.

3.1.3 Manual Screening. To filter relevant information, we manually analyzed the collected reports and posts. We assigned two authors with extensive smart contract development experience to manually filter through 1,379 audit reports and 326 related posts, retaining only content related to *EVM-Inequivalent Code Smells*. **For audit report filtering.** Audit reports include *Issues*, *Root Causes*, and related *Recommendations*. Two authors quickly determined whether the issues were related to EVM-Inequivalent caused by contract reuse by reading these sections. **For Stack Overflow post filtering.** *Stack Overflow* posts also include *Titles*, *Descriptions*, and *Comments*. Two authors quickly identified and removed irrelevant content. Additionally, if the authors could not directly determine whether a post should be retained based on its content, it was categorized as “*tentative*”. Finally, the authors compared results and discussed differences, identifying 197 relevant security analyses of contract reuse on EVM-compatible blockchains.

3.2 Data Analysis

To classify *EVM-Inequivalent Code Smells*, we utilized *Card Sorting*, a user-involved design method, to organize and categorize them without predefined rules [31]. For each security report or *Stack Overflow* post, we created a card with different sections, allowing readers to sort and filter based on their understanding and preferences.

Figure 1 shows an example of the *Stack Overflow* post card, divided into three parts: *Title*, *Description*, and *Comments*. The post’s author wants to deploy ERC-20 tokens using the same contract address on multiple blockchains and seeks a solution. Replies in the *Comments* mention the possibility of deploying the contract with the same address format and deployment method

on *Ethereum* [49] and *Binance Smart Chain* (BSC) [48], but *Tron* network [41] generates addresses differently, making it impossible to deploy a contract with the same address as *Ethereum*. This highlights that even the same contract cannot guarantee the same address across different blockchains due to the varying address generation methods [24, 44]. The post, viewed 3,000 times, highlights the significance of this issue: reused contracts containing calls to fixed-address contracts can be easily exploited for phishing scams on EVM-compatible blockchains. This problem is representative, reproducible, and worthy of attention, summarized as *Phishing Contract Address* (PCA).

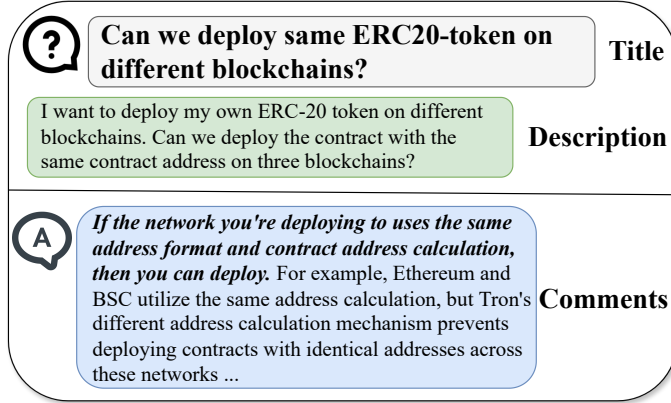


Fig. 1. An example of the *Stack Overflow* Post (SOP).

Figure 2 shows an example of the security audit report card, including three parts: *Issue*, *Root Causes* and *Recommendation*. The report pointed out that the *permit()* function did not strictly follow the *EIP-2612* standard [30], and there was an extra *nonce* parameter in the input parameter. As the *nonce* cannot be specified from external input, the auditors recommended removing the *nonce* parameter in the *permit()* function and removing the statement that verifies the *nonce* to ensure consistency with the *EIP-2612* standard. After discussion, we classified this issue as a *Cross-Chain Replay Attack* (CCRA) due to the incorrect signature verification, which could lead to cross-chain replay attacks, particularly when reusing contracts across different EVM-compatible blockchains.

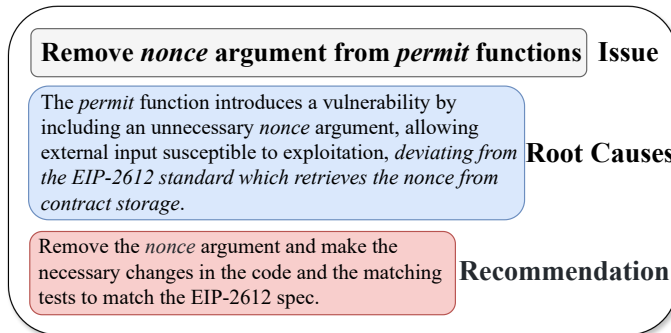


Fig. 2. An example of the Security Audit Report (SAR).

Two authors analyzed 197 collected cards to classify code bad smells in reuse contracts. They examined the root causes, descriptions and comments, placing code smells into existing categories

if possible. If not, they analyzed whether the problems were representative to determine if new categories were needed. The authors compared results, discussed differences, and identified six types of *EVM-Inequivalent Code Smells*.

Table 3. Statistical Distribution of Different Code Smells Sources

Type	CCRA	TDT	PCA	GLI	FGR	BHM
Quantity	82	32	11	38	13	21
Percentage	41.62%	16.24%	5.58%	19.29%	6.60%	10.66%
SOP/SAR	0/82	15/17	11/0	0/38	0/13	0/21

Table 3 provides statistics on the sources of different defect types, showcasing their distribution in *Stack Overflow Posts* (SOPs) and *Security Audit Reports* (SARs). *Cross-Chain Replay Attack* (CCRA) type leads with 82 instances, accounting for 41.62% of the total. *Gas Limit Imbalance* (GLI), *Fixed Gas Reentrancy* (FGR), and *Block Height Misalignment* (BHM) defects are only recorded in SARs, suggesting these problems are more known to professional auditors and less known to the external developer community. In contrast, *Phishing Contract Address* (PCA) defects are only noted in SOPs, indicating these are problems developers encounter and seek help for. *Time Discrepancy Trap* (TDT) defects are collected with 15 SOPs and 17 SARs, highlighting this defect type as a focus in the developer community and in audits. These data reveal differences in the sources of defect types, helping us understand the common issues developers face when reusing contract codes.

3.3 EVM-Inequivalent Code Smells Definition

In this section, we provide definitions of six types of *EVM-Inequivalent Code Smells* in Table 4, followed by comprehensive definitions and illustrative code samples.

Table 4. Definitions of *EVM-Inequivalent Code Smells* in *Ethereum*'s reuse contracts

Code Smell	ID	Definition
<i>Cross-Chain Replay Attack</i>	CCRA	Incorrect chain information validation leads to cross-chain transaction replay.
<i>Time Discrepancy Trap</i>	TDT	The changing block time causes unexpected state updates.
<i>Fixed Gas Reentrancy</i>	FGR	Reentrancy vulnerabilities caused by changes in Gas cost.
<i>Block Height Misalignment</i>	BHM	Different block heights cause inconsistent execution.
<i>Phishing Contract Address</i>	PCA	Stealing permissions from a designated contract address and impersonating it.
<i>Gas Limit Imbalance</i>	GLI	The specified gas limit in the contract leads to denial of service.

(1) Cross-Chain Replay Attack (CCRA). Signatures allow users to authorize others to execute transactions and ensure their legitimacy. Signature verification typically involves validating chain-specific information, such as the *Chain ID*, to prevent signature replay across different blockchains [30]. However, due to a lack of security awareness among developers, many multi-chain contracts lack chain-specific checks during signature verification, leaving them vulnerable. When these contracts are reused, they become vulnerable to replay attacks, where attackers can obtain signatures on *Ethereum* and replay them on other EVM-compatible blockchains.

Code Example: In Figure 3, `verifyEIP712()` function (line 7) validates the signature information (v , r , s) to confirm the signer's authenticity. It utilizes the `ecrecover()` to recover the signer's address and checks if the address matches the target address [36]. The problem stems from the `setter()` function (line 3), allowing anyone to change the `chainId` and `DOMAIN_SEPARATOR` variables, making permission verification easy to bypass. Attackers can exploit the lack of *Chain ID* verification to perform cross-chain signature replay attacks.

```

1 bytes32 public DOMAIN_SEPARATOR;
2 uint256 public chainId;
3 function setter(uint256 _chainId) public {
4     chainId = _chainId;
5     DOMAIN_SEPARATOR = keccak256(abi.encode(keccak256(..., chainId, address(this)
6         ))));
7 }
8 function verifyEIP712(address target, bytes32 hashStruct, uint8 v, bytes32 r,
9     bytes32 s) public view returns (bool) {
10     bytes32 hash = keccak256(abi.encodePacked("\x19\x01", DOMAIN_SEPARATOR,
11         hashStruct));
12     address signer = ecrecover(hash, v, r, s);
13     return(signer == target);
14 }

```

Fig. 3. The example of *Cross-Chain Replay Attack (CCRA)*

(2) **Time Discrepancy Trap (TDT)**. Developers prefer using *block.number* over *block.timestamp* to calculate time intervals, as it is less susceptible to miner manipulation attacks [5]. However, this can lead to *Time Discrepancy Trap (TDT)* when reusing contracts across multiple blockchains due to varying block generation times (e.g., approximately 15 seconds on *Ethereum*). Failing to consider these differences in block generation times when reusing contracts can lead to unexpected results.

```

1 uint256 constant public BLOCKS_PER_WEEK = 43200; // The approximate number of
2   Ethereum blocks per week.
3 function TimeLockedWithdraw() external {
4     require(block.number >= depositBlock[msg.sender] + BLOCKS_PER_WEEK, "Funds
5         are still locked!");
6     balances[msg.sender] = 0;
7     payable(msg.sender).transfer(balances[msg.sender]);
8 }

```

Fig. 4. The example of *Time Discrepancy Trap (TDT)*

Code Example: In Figure 4, the *TimeLockedWithdraw()* function (lines 2–6) calculates time intervals using *BLOCKS_PER_WEEK*, reducing reliance on timestamps and improving cross-chain deployment. However, on EVM-compatible blockchains with faster block generation, the 43,200 block time might not accurately represent a week, impacting calculation accuracy. This inconsistency, due to varying block generation times, can cause the actual withdrawal time to deviate from the intended 1-week, impacting the logic’s reliability and predictability. Attackers could exploit this by triggering early or delayed withdrawals, potentially leading to fund loss or logic violations.

(3) **Phishing Contract Address (PCA)**. To enhance code readability and simplify development, developers often use fixed addresses in contract, such as for Uniswap routers [43], a common practice in certain industries. However, when reusing contracts across multiple chains, these hardcoded addresses can lead *Phishing Contract Address (PCA)*, which attackers can exploit. This vulnerability arises because *Contract Account’s* ownership is determined by contract logic, preventing direct ownership transfer across blockchains. Attackers can leverage the *create()* function to deploy contracts and gain ownership of the generated address on different chains. As the *Wintermute* attack, attackers exploited signature replay and nonce collision techniques to take over target addresses on Optimism [26], impersonating the victim and launching phishing attacks.


```

1  contract SwapToken {
2      // Uniswap Router address on mainnet
3      address RouterAddr = 0x7a250d56...59F2488D;
4      RouterV2 uniswapRouter = RouterV2(RouterAddr);
5      // Exchange ETH for Token in Uniswap
6      function swapEthForTokens(...) external payable {
7          uniswapRouter.swapExactETHForTokens(...);
8      }
9  }

```

Fig. 5. The example of *Phishing Contract Address (PCA)*

Code Example: Figure 5 shows a contract utilizing *Uniswap*'s Router contract address [43] on the *Ethereum* mainnet. While *Uniswap*'s official EOA has permissions on *Ethereum*, these permissions are unknown or undefined on other EVM-compatible blockchains. This opens a window for attackers to deploy a fake Router contract on these alternative chains and exploit the *swapEthForTokens()* function (line 6) to steal ETH and distribute counterfeit tokens. This vulnerability exists because contract ownership cannot be directly transferred across blockchains in the same way as EOAs. As a result, attackers can deploy contracts with identical addresses on different blockchains, enabling sophisticated phishing attacks that exploit these discrepancies.

(4) Gas limit Imbalance (GLI). Gas measures the computational effort needed to execute contracts, preventing blockchain resource abuse. Developers often set a fixed gas limit in contracts. However, setting an excessively high or low gas limit can lead to issues [1]. Due to varying Gas cost standards across different blockchains, a unified gas limit can be easily exploited by attackers. When reusing contracts across blockchains, developers need to dynamically adjust gas limits to ensure successful transactions and avoid wasting gas.

```

1  Payee[500] payees;
2  uint256 nextPayeeIndex;
3  function payOut() public returns (uint) {
4      uint256 i = nextPayeeIndex;
5      while (i < payees.length && gasleft() > 400000) {
6          payees[i].value = 0;
7          payees[i].value = payees[i].value + 1;
8          i++;
9      }
10     nextPayeeIndex = i;
11     return nextPayeeIndex;
12 }

```

Fig. 6. The example of Gas limit imbalance (GLI)

Code Example: The *payOut()* function (lines 3–12) in Figure 6 iterates over the *Payee[]* array, which has a length of 500. It uses the *gasleft()* function (line 5) to ensure that the current remaining gas is greater than 400,000. However, when reusing this code on blockchains with lower gas cost standards, the 400,000 gas requirement can easily cause transactions to fail due to the high limit. This results in the loop exiting prematurely, preventing some elements of the *Payee[]* array from being processed normally and potentially triggering a *Denial of Service (DoS)* attack. This issue underscores the importance of adjusting gas limits to accommodate different blockchain environments, ensuring reliable execution.

(5) Fixed Gas Reentrancy (FGR). The *transfer()* and *send()* functions can effectively prevent Reentrancy attacks on *Ethereum* by limiting the maximum Gas consumption to 2300 units. This limitation restricts the ability to execute additional function calls or alter contract states within the scope of a transaction on *Ethereum*. However, this mechanism may not be consistently implemented across different blockchains, or other platforms may feature lower Gas cost standards [15]. Consequently, when a contract is redeployed on an EVM-compatible blockchain without adjusting for these discrepancies, it may become susceptible to security vulnerabilities.

```

1 function withdraw() external {
2     // transfer(): Send funds, 2300 gas fixed.
3     payable(msg.sender).transfer(balances[msg.sender]);
4     // Update user balance
5     balances[msg.sender] = 0;
6 }

```

Fig. 7. The example of Fixed Gas Reentrancy (FGR)

Code Example: As shown in the *withdraw()* function in Figure 7, the caller is allowed to transfer money through *transfer()*. However, if the caller is a malicious contract, its fallback function may call the *withdraw()* function again when receiving Ether, completing a Reentrancy attack. Although the 2300 gas limit can currently prevent Reentrancy attacks on *Ethereum*, this defense may be ineffective on other blockchains with lower Gas cost standards. To achieve lower transaction fees, the Gas cost of many EVM-compatible blockchains is constantly changing. Therefore, relying on a specific 2300 Gas costs is a vulnerable pattern that cannot fundamentally eliminate the occurrence of Reentrancy vulnerabilities [15].

(6) Block Height Misalignment (BHM). Block height, being a relatively stable metric, is often used by developers in smart contracts for operations like voting, governance, and contract upgrades. However, when reusing these contracts on different EVM-compatible blockchains, each blockchain has its independent block height progression. The specific heights referenced in the code may not match the target blockchain's heights. This mismatch can lead to severe consequences, such as the contract logic failing to execute intended operations at desired block heights or causing unintended consequences due to differences in block height progression between blockchains [6]. Attackers can exploit this, potentially leading to fund loss, governance issues, or critical failures.

```

1 // Hard fork for DAO
2 uint constant DAO_FORK_BLOCK = 1760000;
3 function handleFork() public {
4     if (block.number >= DAO_FORK_BLOCK) {
5         ...
6     }

```

Fig. 8. The example of Block Height Misalignment (BHM)

Code Example: The *handleFork()* function in Figure 8 is designed to handle the DAO fork on *Ethereum* by setting a specific block height, *DAO_FORK_BLOCK* (at 1760000), to identify the event's occurrence. The function compares the current block height with the DAO fork's block height to execute the processing logic. An overly large block height requirement could trigger a *Denial of Service* attack, or changes in block height progression may cause unintended executions. Therefore, it is crucial to carefully consider using fixed block heights when reusing smart contract code on different EVM-compatible blockchains to ensure robustness and security.

4 Methodology

4.1 Overview

The workflow of *EquivGuard* is shown in Figure 9. It takes the *Ethereum* Solidity contract source code as input because *Ethereum* contracts are most likely reused by other Blockchains. The detection process consists of three main steps: generating *Inter-contract Program Dependency Graph* (I-PDG) [46] and the *Control Flow Graph* (CFG) [28], identifying suspicious paths through static taint analysis, and verifying path feasibility via symbolic execution. **Step 1: I-PDG and CFG Generation.** *EquivGuard* analyzes the contract source code to generate two graphs, which serve as inputs for Step 2 and Step 3. **Step 2: Domain-Guided Static Taint Analysis.** We utilize different *Domain-Specific Patterns* to guide taint propagation analysis, identify and track potentially paths. Combining different exploitation modes, key instructions, and reverse analysis enables efficient path search. **Step 3: Symbolic Execution Verification.** Based on the suspicious path information, *EquivGuard* verifies the reachability of the path through symbolic execution to improve the accuracy of code smell detection. Finally, *EquivGuard* outputs the detection results.

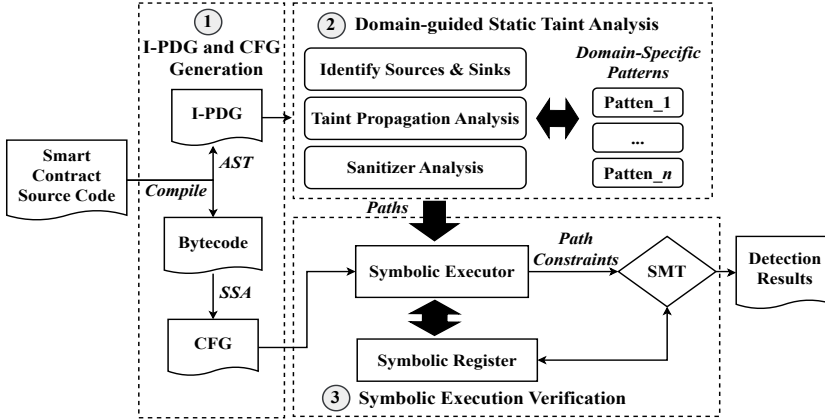


Fig. 9. The workflow of *EquivGuard*.

4.2 I-PDG and CFG Generation

To model and analyze contract semantics, *EquivGuard* compiles the source code to obtain the *Abstract Syntax Tree* (AST) and bytecode. It then constructs the *Inter-contract Program Dependency Graph* (I-PDG) [46] and the *Control Flow Graph* (CFG) [28], which capture program dependencies and control flow, respectively. These graphs serve as inputs for Step 2 and Step 3 of the process.

The AST provides rich syntactic information about the code, including the positions and relationships of keywords, variables, and instructions. This information aids in the global dependency analysis and construction of the I-PDG, which is generated by integrating program dependency analysis between AST statement blocks into the *Inter-contract Control Flow Graph* (I-CFG) [29]. The I-PDG contains the global program dependencies of the contract and systematically analyzes the program semantics of smart contracts. With accurate data and control dependencies, the I-PDG facilitates efficient data flow tracking, identification of potential logical errors, and analysis of special semantics. For example, when detecting *Cross-Chain Replay Attack* (CCRA), the *recover()* function is critical. *EquivGuard* uses AST analysis to directly determine the dependency path set

from the external function entry to the *ecrecover()* function call instruction, combining static taint analysis between key instructions to identify dangerous vulnerabilities.

While SSA facilitates the analysis of underlying execution details, its lack of semantic context makes it ineffective for the global syntactic analysis of contracts. The CFG generated from bytecode is crucial for symbolic execution path traversal. During CFG path recovery, we convert stack operations in the bytecode to SSA form and perform constant propagation analysis to solve the dynamic jump address calculation problem. By converting the bytecode to *Static Single Assignment* (SSA) form [46], we ensure complete path recovery, enabling the symbolic executor to collect comprehensive path constraints.

4.3 Domain-guided Static Taint Analysis

As smart contracts become larger and more complex, involving intricate function calls, contract inheritance, and cross-contract calls, among others, the growth of path states has exploded. Heuristic and coarse-grained data flow analysis can easily lead to overfitting problems, seriously affecting the accuracy of results. To address these challenges, we propose data flow analysis and path reverse analysis based on key instructions to achieve an efficient path search for static taint analysis.

(1) Data flow analysis between key instructions. *EquivGuard* utilizes I-PDG with key instructions to implement data flow analysis between global instructions. The I-PDG converts global function calls and cross-contract calls into code block² jumps in the graph, ensuring correct processing of return values. This helps identify patterns and perform sanitizer analysis by revealing dependencies between contract variables, improving static taint analysis accuracy and efficiency. At the same time, *EquivGuard* identifies key instructions as sinks, marks critical variables influenced by external inputs as sources, and performs taint propagation analysis based on the I-PDG while conducting sanitizer analysis. *EquivGuard* avoids traversing irrelevant programs by analyzing paths between key instructions of patterns, thereby improving the accuracy and flexibility of the analysis. The selection of key instructions depends on different *Domain-Specific Patterns*. As shown in Figure 10, for static taint analysis of *Cross-Chain Replay Attack* (CCRA), *EquivGuard* uses the *ecrecover()* call instruction as *Taint* and external inputs as *Source*. By obtaining two different paths from the dependency relationship between the key instructions of *Source* and *Taint*, further data flow analysis is achieved. The *ecrecover()* function is a key function, and *EquivGuard* uses key instruction guidance to ensure efficient path search. For more detailed detection methods, please refer to section 4.5.

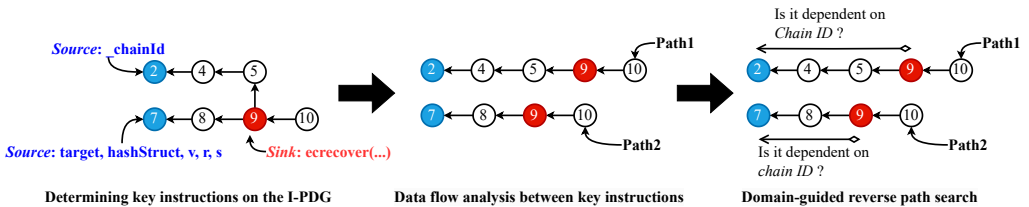


Fig. 10. Domain-Guided Static Taint Analysis for *Cross-Chain Replay Attack* (CCRA). The numbers in the circles are the line numbers from Figure 5. Blue circles are *Source* points, red circles are *Sink* points, and arrows show program dependencies.

²Code block refers to a sequence of instructions in the contract's control flow that are executed sequentially without branching or function calls.

(2) Domain-guided reverse path search. To prune paths, *EquivGuard* starts from the variable usage statement in the critical function and traces it backward to the assignment or declaration point of the variable. Therefore during static taint analysis, starting from the taint sinks, the data flow is traced back to its sources. Complex function calls and cross-contract calls involved are analyzed through I-PDG, identifying any sanitizer mechanisms in the data flow. Based on the I-PDG's data flow analysis, *EquivGuard* effectively focuses on states and value sources when the taint sinks are used.

Since *EVM-Inequivalent Code Smells* are closely related to the incorrect dependence of the reused contract on the specific blockchain state, the main challenge is to analyze whether the variable's value remains unchanged throughout the execution without running the program code. As shown in Figure 10, *EquivGuard* starts from the taint sinks on different paths and uses dependency relationships to reverse analyze any dynamic dependence on *blockchainID* (from the blockchain environment). *EquivGuard* combines I-PDG with reverse analysis of critical functions to analyze variable propagation and detect smells. This reverse tracing helps analyze any contamination relationship between function input and output and whether the data has been verified, cleaned, or protected, achieving efficient attribute verification of paths.

4.4 Symbolic Execution Verification

To improve detection accuracy, *EquivGuard* employs symbolic execution to traverse suspicious paths generated in *Step 2* and verify their feasibility. To mitigate the effect of calling permissions determined by specific rules that may affect path analysis and result in false positives, we introduce symbolic execution to collect path constraints and verify the reachability of suspicious paths, thereby enhancing the accuracy of code smell detection. *EquivGuard* traverses the CFG branch to model the contract's storage and collect path constraints, then proves reachability through path constraint solving. Therefore, we introduce the *Symbolic Register* to store symbolic states and path constraint information, overcoming the challenge of incomplete path constraint collection. By analyzing the state storage operation of bytecode and combining it with the *Z3 solver*, we store specific values and symbolic expressions as key-value pairs in the *Symbolic Register*.

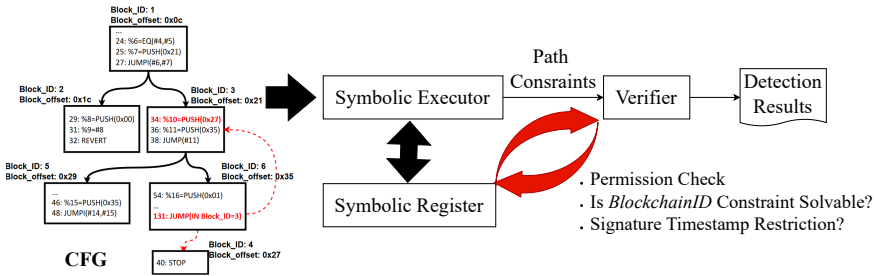


Fig. 11. Symbolic Execution Verification for *Cross-Chain Replay Attack* (CCRA).

As shown in Figure 11 for the symbolic execution verification targeting *Cross-Chain Replay Attack* (CCRA), the *Symbolic Executor* traverses paths and collects path constraints. The *Verifier* queries the *Symbolic Register* to prove whether the *BlockchainID* Constraint is solvable, whether there are *Signature Timestamp Restrictions*, and whether there is permission verification of the caller's identity. Solving these constraints ensures the reachability of paths and completes attribute verification. This symbolic modeling is key to a complete collection of path constraints, which can be used to check whether the contract address involved in the external contract function call is symbolic. Based on this, we determine whether the contract call may be externally controlled, effectively identifying

potential function callback vulnerabilities. The completed path also supports cross-contract path constraint analysis, enhancing the security verification of contracts.

4.5 EVM-Inequivalent Code Smells Detection

In this section, we detail the *Domain-Specific Patterns* required for detection and how to guide the path search:

(1) **Cross-Chain Replay Attack (CCRA).** *EquivGuard* checks if the contract allows the logic of the function to be approved and executed by external parties via signatures, such as *permit()* function. Furthermore, it examines whether the signature verification process includes *Chain ID* and blockchain information validation. By locating calls to the *ecrecover()* function within the contract, *EquivGuard* can narrow down the scope of function search. Coupled with reverse taint analysis, it determines if the variable related to *Chain ID* can be easily manipulated in the code. Symbolic execution is then used to verify path feasibility for accurate detection.

(2) **Time Discrepancy Trap (TDT).** *EquivGuard* analyzes whether the contract employs block generation numbers to calculate time consumption by locating code blocks involving *block.number*. It performs reverse taint analysis to identify the sources and propagation paths of variables. *EquivGuard* then determines if these variables are constants and examines the judgment conditions of the calculation results. Finally, symbolic execution verifies the feasibility of the identified path and potential inconsistencies in time calculation across different EVM-compatible blockchains, ensuring accurate detection.

(3) **Phishing Contract Address (PCA).** *EquivGuard* analyzes whether contracts contain unrestricted method calls to fixed contract addresses. To reduce false positives, *EquivGuard* employs static taint analysis to track the propagation of these addresses within the contract. It uses symbolic execution to verify the feasibility of paths involving such calls.

(4) **Gas Limit Imbalance (GLI).** *EquivGuard* analyzes contracts to detect operations setting specific gas limits. Specifically, there is a query for the remaining gas amount using the *gasleft()* function [2] and adjusts the logic accordingly. By employing I-PDG combined with static taint analysis, it determines whether such logic exists in the code and whether the restriction is subject to conditional checks. If such conditions are found, GLI is flagged.

(5) **Fixed Gas Reentrancy (FGR).** *EquivGuard* first analyzes whether the contract includes token transfers caused by the *transfer()/send()* methods. Second, it examines whether the contract follows the safe development pattern of *Check->Effect->Interaction* (C-E-I)³ [9, 47]. If the contract includes *transfer()/send()* methods but does not adhere to the safe *Check->Effect->Interaction* development pattern, then *EquivGuard* considers this smell to exist.

(6) **Block Height Misalignment (BHM).** *EquivGuard* analyzes whether the contract contains restrictions on specific block heights. It utilizes I-PDG to examine whether the contract's execution statements impose restrictions on specific block heights. Additionally, it performs taint analysis to track the propagation of variables related to block heights and determine their impact on control flow decisions. Furthermore, *EquivGuard* employs symbolic execution to verify the feasibility of the identified paths that involve block height restrictions, ensuring accurate detection.

5 Evaluation

In this section, we analyze and evaluate the effectiveness of *EquivGuard* in detecting *EVM-Inequivalent Code Smells* by answering the following research questions:

- RQ1: How does *EquivGuard* perform on real large-scale datasets?

³C-E-I requires ensuring timely state updates before interacting with external contracts, reducing the possibility of malicious contracts attempting to hijack control flow after external calls.

- RQ2: What is the performance of *EquivGuard* in detecting *EVM-Inequivalent Code Smells*?
- RQ3: What is the impact of different phases within *EquivGuard*?

Experimental Setup. The experiment was conducted on a computer running Ubuntu 20.04 LTS, equipped with a 16-core Intel(R) Xeon(R) Gold 5217 processor and 120 GB of memory. We use `solc-select` [17] to switch compiler versions and query the asset through *Etherscan*'s API [20].

5.1 RQ1: Performance on large-scale datasets

Dataset. To analyze the performance of *EquivGuard* on large-scale datasets and explore the distribution of *EVM Inequivalent Code Smells* across different blockchain environments, we collected and experimented with 905,948 contract source codes from six mainstream EVM-compatible blockchains. These blockchains account for 80.64% of the total crypto-market assets by the time of writing the paper [19], including *Ethereum* (66.25%), *Binance Smart Chain* (BSC) (6.23%), *Arbitrum* (4.29%), *Polygon* (1.35%), *Optimism* (1.26%), and *Avalanche* (1.26%).

Results. Table 5 shows the statistical results of *EquivGuard*'s detection on large-scale datasets across different blockchains. Overall, *EVM-Inequivalent Code Smells* are widely distributed across various blockchains. 115,148 contracts on the *Binance Smart Chain* (BSC) contain code smells, accounting for 31.80%. *Avalanche* follows with a proportion of 21.62%, while *Optimism* has the lowest proportion at 10.08%. Among the various types of code smells, *Phishing Contract Address* (PCA), *Time Discrepancy Trap* (TDT), and *Fixed Gas Reentrancy* (FGR) have the highest proportions, especially on the *Binance Smart Chain* (BSC), where each of these three types accounts for nearly 10%. *Avalanche* shows a significant presence of *Time Discrepancy Trap* (TDT) (2,333 cases, 6.37%) and *Fixed Gas Reentrancy* (FGR) (3,658 cases, 9.98%). In contrast, *Block Height Misalignment* (BHM) and *Gas Limit Imbalance* (GLI) have proportions below 0.23% across the six blockchains, indicating a general awareness among developers to avoid dependence on specific block height and block time. Overall, the average proportion of *EVM Inequivalent Code Smells* across all blockchains is 17.70%. These data suggest that *EVM Inequivalent Code Smells* are prevalent in different blockchain environments, highlighting developers' lack of awareness regarding these smells in reused contracts and the urgent need to strengthen preventive measures.

Table 5. Statistics of detection results of *EquivGuard* in large-scale datasets

Blockchain	CCRA	PCA	BHM	TDT	GLI	FGR	Total
Arbitrum	771(1.26%)	3152(5.16%)	15(0.02%)	1972(3.23%)	32(0.05%)	2363(3.87%)	8305(13.59%)
Avalanche	443(1.21%)	1441(3.93%)	36(0.10%)	2333(6.37%)	10(0.03%)	3658(9.98%)	7921(21.62%)
BSC	1512(0.42%)	35018(9.62%)	841(0.23%)	39924(10.97%)	43(0.01%)	37810(10.55%)	115148(31.80%)
Optimism	210(1.63%)	593(4.60%)	3(0.02%)	263(2.04%)	13(0.10%)	218(1.69%)	1300(10.08%)
Polygon	2773(1.92%)	7102(4.93%)	183(0.13%)	4219(2.93%)	59(0.04%)	6002(4.17%)	20338(14.12%)
Ethereum	1073(0.37%)	12706(4.42%)	138(0.05%)	5277(1.84%)	16(0.01%)	23955(8.34%)	43165(15.03%)

Answer to RQ1: Analysis of 905,948 contracts across six blockchains revealed the widespread presence of *EVM Inequivalent Code Smells*, averaging 17.70%. *Phishing Contract Address* (PCA), *Time Discrepancy Trap* (TDT), and *Fixed Gas Reentrancy* (FGR) are particularly prominent on *Binance Smart Chain* (BSC) and *Avalanche*. The high prevalence of affected contracts underscores a major security threat and necessitates urgent developer risk prevention.

5.2 RQ2: Evaluation of detection effects

Dataset. To evaluate *EquivGuard*'s performance in detecting *EVM-Inequivalent Code Smells*, we randomly sampled and manually analyzed false positives and false negatives from the detection results to assess *Precision* and *Recall*. Following previous research [55], we used a confidence

interval-based sampling method with a 95% confidence level and a 10-point confidence interval, resulting in 444 positives and 96 negatives from the detection results. Then, we manually checked these samples to determine the existence of smells, ultimately obtaining 426 true positive samples and 114 true negative samples. Table 6 presents the number of samples for positive label contracts with different *EVM-Inequivalent Code Smells*.

Table 6. Precision Analysis Results Statistics of *EquivGuard*

	PCA	GLI	CCRA	FGR	TDT	BHM
Samples	95	14	88	96	94	57
TP	95	14	86	91	88	48
FP	0	0	2	5	6	9
Precision	100.00%	100.00%	97.73%	94.79%	93.62%	84.21%

5.2.1 False Positive Analysis. To analyze the precision of *EquivGuard*, we randomly sampled the positive detection results according to different code smell categories and performed a false positive analysis. For each item, we calculate precision through the formula $Precision = TP / (TP + FP)$, and calculate the overall detection $Precision_{(Overall)} = \frac{\sum_{i=1}^n p_{c_i} \times |c_i|}{\sum_{i=1}^n |c_i|}$, p_{c_i} denotes the precision for code smell i , and $|c_i|$ represents the number of code smell i [54].

Results. According to the results in Table 6, the $Precision_{(Overall)}$ of *EquivGuard* reached 95.29%, indicating that it has high accuracy in detecting *EVM-Inequivalent Code Smells*. Through result analysis, we found that some false positives are related to the variability of permission control. Customized and diverse permission control is a key way to ensure contract security, but it also increases the complexity of analysis. For example, in the *Block Height Misalignment* (BHM) detection, false positives mainly stem from identifying the key function calling permission. To reduce such problems as possible, we referred to the existing *Path Protection Technologies* (PPTs) [53], but these technologies rely on static rule analysis, only cover the most common permission paths, and cannot guarantee the analysis of all situations. Furthermore, a small number of false positives are related to heuristic analysis invariants. To analyze contract invariants, we adopted a heuristic method of statically analyzing the read-and-write relationships of variables. However, this heuristic method will lead to inaccurate detection results and false positives when dealing with complex data structures such as dynamic arrays.

5.2.2 False Negative Analysis. To evaluate the reliability of *EquivGuard*'s detection, we calculate recall using $Recall = TP / (TP + FN)$ and perform cause analysis.

Results. Through manual inspection and analysis, the detection recall rate was calculated to be 99.06%. We found 4 undetected (false negatives) code smells, all of which were *Cross-Chain Replay Attack* (CCRA). These code smells mainly occur when calling the *ecrecover()* for signature verification, which is implemented using assembly code. As EVM-compatible chains have limited support for opcodes and precompiled instructions, which restricts the scope for reusing contracts with assembly code. Research [11] shows that manual inline assembly lacks generality, and the intricate logic raises developers' comprehension costs, making direct reuse less favorable. Furthermore, we plan to expand assembly code analysis capabilities in our subsequent work.

Answer to RQ2: *EquivGuard* achieved 95.29% Precision and 99.06% Recall in *EVM-Inequivalent Code Smells* detection, demonstrating high reliability.

5.3 RQ3: Ablation Experiment

Dataset and Group Settings. To evaluate the necessity of Step 2's *Domain-guided Reverse Path Search* (DRPS) and Step 3's *Symbolic Execution Verification* (SEV) in *EquivGuard*, we conducted a

series of ablation experiments with four groups: **Neither** (both DRPS and SEV disabled), **No Path Guidance** (DRPS disabled), **No Symbolic Execution** (SEV disabled), and **Full Setup** (both DRPS and SEV enabled), with a unified time limit of 60 seconds. To ensure accuracy and reliability, we used the RQ2 dataset, which includes 426 positive samples and 114 negative samples, manually checked and analyzed to ensure the validity of the results.

5.3.1 Effectiveness Analysis. In the four experimental groups, the implementation of different functional components resulted in varying detection capabilities, each with distinct characteristics. The **Full Setup** group, where both DRPS and SEV were enabled, achieved the highest precision (95.05%) and recall (99.06%), indicating that the combination of domain-guided path search and symbolic execution path validation can achieve accurate and reliable detection. In contrast, disabling SEV in the **No Symbolic Execution** group resulted in a significant drop in recall (41.08%) and precision (65.30%), demonstrating the importance of symbolic execution in enhancing true positives and reducing false negatives. Similarly, disabling DRPS in the **No Path Guidance** group led to a substantial decline in recall (9.86%) and precision (27.81%), highlighting the crucial role of domain-guided path search in guiding analysis and reducing false positives. Finally, the **Neither** group, which disabled both DRPS and SEV, performed the worst, with precision (11.20%) and recall (3.29%) dropping to extremely low levels, further proving the necessity of domain-guided path search and symbolic execution path validation.

Table 7. Ablation Study Results

Metric	Full Setup	No Symbolic Execution	No Path Guidance	Neither
# TP	422	175	42	14
# FP	22	93	109	111
# FN	4	251	384	412
Precision (%)	95.05%	65.30%	27.81%	11.20%
Recall (%)	99.06%	41.08%	9.86%	3.29%
Average Time (s)	45.2 s	21.1 s	57.2 s	10.3 s
# Timeouts	4 (0.74%)	0 (0)	336 (62.22%)	0 (0)

5.3.2 Efficiency Analysis. We also analyzed the average time consumption under different experimental groups and found that *EquivGuard*'s phased strategy helps combine the characteristics of different techniques to ensure efficient detection. The **Full Setup** group had an average time of 45.2 seconds, which is moderate. Although the **No Symbolic Execution** group had an average time of only 21.1 seconds, mainly due to reduced time for symbolic execution path traversal and constraint solving, this also resulted in a significant decline in detection effectiveness. The **No Path Guidance** group had the longest average time (57.2 seconds) and the highest number of timeouts (336), indicating that Step 2's *Domain-guided Reverse Path Search* (DRPS) is crucial for alleviating the intensive computation and path explosion faced by symbolic execution path reachability verification. Although the **Neither** group had the shortest average time (10.3 seconds), its detection effectiveness was extremely poor.

5.3.3 Analysis of the Impact of Symbolic Execution Timeout. Timeouts significantly impact the efficiency of symbolic execution, thus affecting the performance of *EquivGuard*. Path explosion and blind path traversal can easily cause timeouts, limiting the scope of path exploration, reducing detection accuracy, and hindering scalability. As shown in Table 7, by comparing the **Full Setup** and **No Symbolic Execution** groups, it is evident that the introduction of symbolic execution significantly improves recall from 41.08% to 99.06%, and also enhances precision. Meanwhile, in the

absence of path guidance during the symbolic execution's path reachability verification (**No Path Guidance**), detection precision and recall drop to 27.81% and 9.86%, respectively. Therefore, blind path traversal and path explosion can easily lead to timeouts. This not only limits the paths explored by symbolic execution but also may mistakenly identify unverified paths as issues due to interrupted verification, significantly increasing false positives. Additionally, the **No Path Guidance** group had the longest average analysis time (57.2 seconds) and the most timeouts (336), severely affecting efficiency. Frequent timeouts also expose scalability issues of *EquivGuard* when handling large and complex programs, limiting its practical applicability. Our analysis of experimental results identified three main causes of timeouts: path explosion, constraint-solving complexity, and external dependencies. For a detailed analysis, please refer to Appendix A.

Answer to RQ3: The analysis shows that *Domain-guided Reverse Path Search* (DRPS) aids in the traversal of paths by symbolic execution, while *Symbolic Execution Verification* (SEV) effectively validates path reachability, improving both precision and recall. *EquivGuard* combines the characteristics of different techniques to achieve efficient detection.

6 Discussion

6.1 Exploiting Cross-Chain Replay Attack (CCRA) in Multichain Project

Through large-scale detection, *EquivGuard* discovered that *Multichain*'s cross-chain project is affected by *Cross-Chain Replay Attack* (CCRA), compromising the security of its managed assets. In this subsection, we will analyze how the *Cross-Chain Replay Attack* (CCRA) in the reused contract is exploited and the resulting asset risks. *Multichain*'s contract contains the *Cross-Chain Replay Attack* (CCRA), which arose from hard-coding the chainId as 122⁴, while the actual deployed blockchain's chainId was 1284. This caused the failure of the verification mechanism intended to prevent replay attacks. This threat continuously affects assets on other chains where the same contracts are reused.

The overall exploitation process consists of three stages: (I) Retrieve signatures with *ChainID* 122. Hackers can search for their previously used signature information from historical transactions of a blockchain with *ChainID* 122 or create related signature information offline. (II) Verify signatures on the chain with *ChainID* 1284. Hackers call the *transferWithPermit()* function on the *AnyswapV5ERC20* contract deployed on blockchain with *ChainID* 1284 using the signatures. The core reason why the signature verification succeeds is due to a check using the incorrect *ChainID* 122 in the signature verification logic. (III) Profit. Attackers transfer tokens to gain profit.

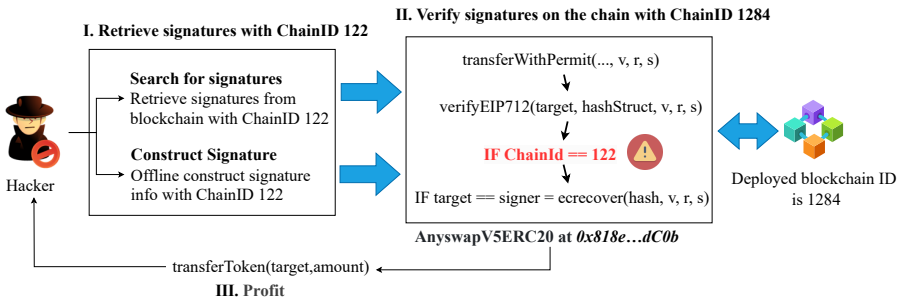


Fig. 12. CCRA Exploitation in the *Multichain*'s contract.

⁴<https://moonscan.io/address/0x818ec0a7fe18ff94269904fced6ae3dae6d6dc0b#code#L306>

Attackers can search for exploitable historical transaction signatures and reuse them across different chains to generate fraudulent transactions. As *Multichain*'s cross-chain router at `0x818e...dC0b` used this contract code, approximately \$28.52 K in assets were put at risk. *EquivGuard* promptly identified and reported this code smell.

6.2 Implications

EVM-inequivalent code smells expose design flaws in contract reuse, leading to contract execution inconsistency. *EquivGuard* automatically detects these code smells before reusing, improving the reliability and reusability of reused code. This study of *EVM-inequivalent code smells* and *EquivGuard* may provide new insights for secure contract reuse.

For Auditors. Auditors can use *EquivGuard* to mitigate risks from *EVM-inequivalent code smells*. While code reuse enhances audit efficiency, it introduces new attack vectors when reused across blockchains. For instance, *Wintermute*'s wallet attack [26] resulted in a \$20 million loss when Ethereum-safe code was reused on *Optimism*. *EquivGuard* helps auditors analyze multi-chain execution inconsistencies in reused contracts and provide comprehensive security guidance based on different blockchain designs (e.g., *Gas Mechanisms* and *Consensus Protocols*).

For Developers. *EquivGuard* helps developers avoid negative impacts due to the misunderstandings of *directly copying smart contracts for multi-chain deployments*. While multi-chain reuse saves costs, it can introduce *EVM-inequivalent code smells*, creating security threats. *EquivGuard* alerts developers to these code smells, providing location and category information. This serves as a reminder for developers to conduct thorough testing in various execution environments to mitigate the risks associated with EVM-inequivalent execution.

For Researchers. Researchers may further investigate programming practices that cause inconsistent execution in reused contracts to provide more comprehensive advice for development. For instance, they could develop a tool to automatically identify the negative impacts of *EVM-inequivalent code smells*, offering users targeted improvement suggestions.

For Community. The blockchain community can improve the execution layer of blockchains to mitigate risks. This could involve conducting a comprehensive analysis of the root causes of inconsistent execution stemming from reused contracts, considering different *Gas Mechanisms* and *Consensus Protocols*. Furthermore, it could involve implementing version updates to prevent the deployment of problematic contracts or to alert contract deployers.

6.3 Threats to Validity

Internal Validity. One internal threat is that not all code smells necessarily lead to financial loss or attacks; however, they indicate underlying design flaws that can create opportunities for vulnerabilities, especially in new scenarios from multi-chain contract reuse. Developers urgently need to enhance their awareness of *EVM-Inequivalent Code Smells*, especially when reusing contracts across multiple chains. The other internal threat is using *Domain-Specific Patterns* to guide path searches. These patterns, derived from security development experience, are crucial for taint propagation analysis. For instance, in *Cross-Chain Replay Attack* (CCRA) detection, *ecrecover()* calls are identified as sinks, as they are essential for signature verification. The analysis traces taint flow from external inputs (sources) to these sinks, incorporating *chain ID* checks. They provide necessary instructions to help identify sources and sinks and indicate code smell presence.

External Validity. One external threat is the generalizability of the code smell definition. To ensure the identified smells reflect real developer issues, we analyzed 1,379 security audit reports and 326 *Stack Overflow* posts related to contract reuse. This work is the first to define and detect *EVM-Inequivalent Code Smells*, tracing the identified smells back to their sources in *Stack Overflow* and security audits. Another external threat is the risk of incomplete coverage, as using a single tag may

result in missing relevant posts or including irrelevant ones. To ensure data completeness, we used an extended combination of related tags, including “EVM”, “EVM Equivalent”, “EVM Compatible”. This broader tag selection helped capture more relevant posts and minimize omissions. Two rounds of manual reviews were conducted to eliminate irrelevant content.

7 Related Work

7.1 Defining and Detecting Bugs in Smart Contracts

Based on both real-world attacks and theoretical research, researchers have proposed many solutions to address the proliferation of bugs. For example, Luu et al. [28] combined four vulnerability patterns and proposed the symbolic execution detection tool *Oyente* based on bytecode analysis, providing a classic solution for contract vulnerability detection. By analyzing posts on *Ethereum StackExchange* [23] and real-world smart contracts, Chen et al. summarized 20 types of smart contract defects and highlighted 5 high-risk defects caused by protocol errors, providing developers with identification and guidance on fixing contract deficiencies [13]. Zhang et al. [55] through large-scale analysis of *Ethereum* transactions, smart contracts, and *StackExchange* posts, found that developers face 5 main types of obstacles when dealing with crypto-related tasks and surveyed industry insiders to reveal the root causes of these obstacles and suggest improvements, providing practical guidance to improve the encryption task development experience. Yang et al. [54] defined and explained 5 common defect types in *Non-Fungible Token* (NFT) contracts by analyzing *StackOverflow* posts and proposed a symbolic execution-based tool, *NFTGuard*, to automatically detect these using contract *Abstract Syntax Tree* (AST) and bytecode features.

7.2 Research on Smart Contract Reuse

Chen et al. [14] analyzed 146,452 open-source *Ethereum* contracts, finding widespread code reuse with ERC20 tokens being the most commonly reused. Sun et al. [38] studied over 350,000 contracts, observing that 50% of self-developed sub-contracts had duplicate functions, while 35% of external sub-contracts had issues such as inconsistent usage. They also extracted 61 frequent reuse patterns to guide secure contract development. Pierro et al. [34] compared mainstream smart contract corpora, noting that only a small portion reused code from the secure *OpenZeppelin* repository [32]. They recommended leveraging the *OpenZeppelin* Solidity Library to improve contract security. Huang et al. [25] constructed a semantic *Code Knowledge Graph* to uncover unknown factors in smart contract reuse, effectively enhancing code recommendation accuracy and developer efficiency.

Differences. First, this is the first study to focus on the inconsistent execution of reused contracts across EVM-compatible blockchains. While reusing *Ethereum* contracts on different platforms introduces new attack vectors, research on *EVM-Inequivalent Code Smells* has been lacking. Secondly, our analysis combines insights from *StackOverflow* posts and security audit reports from security companies, enabling us to better collect and analyze real-world cases. Thirdly, to detect *EVM-Inequivalent Code Smells*, we propose a new method that combines dynamic and static techniques. This approach aims to achieve more efficient detection by leveraging the efficient path search of static taint analysis and the verification capabilities of symbolic execution, thereby improving the comprehensiveness and accuracy of the detection process.

8 Conclusion

In this paper, we conduct the analysis of *EVM-Inequivalent Code Smells*, which is the first study on inconsistent execution of *Ethereum* contracts when reused on EVM-compatible blockchains. By analyzing 1,379 security audit reports and 326 *Stack Overflow* posts, we identified and defined six *EVM-Inequivalent Code Smells*. Discovering these code smells, which cause inconsistent execution of reused contracts on multi-chains, ensures more secure and reliable contract reuse. To aid in

discovering reused contracts that contain *EVM-Inequivalent Code Smells* in the real world, we designed *EquivGuard* to implement automated detection of contracts. *EquivGuard* identifies key paths by leveraging *Domain-Specific Patterns*, which encapsulate domain knowledge for analyzing various code smells, and combines symbolic execution to verify path reachability. We conducted large-scale experiments on 905,948 contracts across six mainstream blockchains, achieving a detection precision of 95.29%. Additionally, our experiments revealed that 17.70% of contracts contained at least one such code smell. This highlights the prevalence of these code smells and reminds developers to prevent inconsistencies when reusing smart contracts.

9 Data Availability

We have anonymized and made *EquivGuard*'s source code, empirical study materials, experimental datasets, and results publicly available at <https://anonymous.4open.science/r/EquivGuard-68B0>.

Acknowledgments

This research is supported by the National Key Research and Development Program of China (2022YFB2703204), the National Natural Science Foundation of China (No. 62032025, No. 62302534), the Guangdong Basic and Applied Basic Research Foundation (No. 2025A1515011632), and the Major Key Project of Peng Cheng Laboratory under Grant PCL2023A05-2.

References

- [1] 0x52. 2024. An attacker can lock operator out of the pod by setting gas limit that's higher than the block gas limit of dest chain. <https://solodit.xyz/issues/h-01-an-attacker-can-lock-operator-out-of-the-pod-by-setting-gas-limit-thats-higher-than-the-block-gas-limit-of-dest-chain-code4rena-holograph-holograph-contest-git>.
- [2] Solidity Academy. 2023. Understanding Ethereum Gas: A Technical Deep Dive into gasleft(). <https://medium.com/@solidity101/understanding-ethereum-gas-a-technical-deep-dive-into-gasleft-b0842742fd12>.
- [3] Alex Beregszaszi (@axic), Jacques Wagoner (@jacqueswww). 2018. EIP-1380: Reduced gas cost for call to self. <https://eips.ethereum.org/EIPS/eip-1380>.
- [4] Arbitrum. 2024. Arbitrum — The Future of Ethereum. <https://arbitrum.io/>.
- [5] AuditBase. 2024. Consider using block.number instead of block.timestamp. <https://detectors.auditbase.com/blocknumber-vs-timestamp-solidity>.
- [6] AuditOne. 2024. Lack of Validation for valid_till_block_height on FastBridge Service. https://solodit.xyz/issues/lack-of-validation-for-valid_till_block_height-on-fastbridge-service-auditone-none-aurorafastbridge-markdown.
- [7] BlockSec. 2024. Ensuring a Secure and Seamless Web3 World. <https://blocksec.com/>.
- [8] BNB Chain community. 2024. About the BEP Category. <https://forum.bnbchain.org/t/about-the-bep-category/624>.
- [9] CaptPython. 2019. Design pattern Checks-Effects-Interactions Pattern. <https://ethereum.stackexchange.com/questions/66456/design-pattern-checks-effects-interactions-pattern>.
- [10] CFI Team. 2024. Hard Forks. <https://corporatefinanceinstitute.com/resources/cryptocurrency/hard-fork/>.
- [11] Stefanos Chaliasos, Arthur Gervais, and Benjamin Livshits. 2022. A study of inline assembly in solidity smart contracts. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 165 (Oct. 2022), 27 pages. <https://doi.org/10.1145/3563328>
- [12] Che Kohler. 2022. What Is Bitcoin Block Time? <https://thebitcoinmanual.com/articles/btc-block-time/>.
- [13] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2022. Defining Smart Contract Defects on Ethereum. *IEEE Transactions on Software Engineering* 48, 1 (2022), 327–345. <https://doi.org/10.1109/TSE.2020.2989002>
- [14] Xiangping Chen, Peiyong Liao, Yixin Zhang, Yuan Huang, and Zibin Zheng. 2021. Understanding Code Reuse in Smart Contracts. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 470–479. <https://doi.org/10.1109/SANER50967.2021.00050>
- [15] Consensys. 2019. Stop Using Solidity's transfer() Now. <https://consensys.io/diligence/blog/2019/09/stop-using-soliditys-transfer-now/>.
- [16] ConsenSys. 2024. Consensys - A complete suite of trusted products to build anything in web3. <https://consensys.io/>.
- [17] Crytic. 2024. Manage and switch between Solidity compiler versions. <https://github.com/crytic/solc-select>.
- [18] Cyfrin. 2025. Solodit. <https://solodit.cyfrin.io/>.
- [19] DefiLlama. 2024. DefiLlama EVM Chains. Retrieved from <https://defillama.com/chains/EVM>.
- [20] Etherscan. 2024. Etherscan APIs- Ethereum (ETH) API Provider. <https://etherscan.io/apis>.
- [21] Etherscan. 2025. Etherscan Smart Contracts Audit and Security. https://etherscan.io/directory/Smart_Contracts/Smart_Contracts_Audit_And_Security.

- [22] evm.storage. 2024. An Ethereum Virtual Machine Opcodes Interactive Reference. <https://www.evm.codes/>.
- [23] Ethereum Stack Exchange. 2024. Ethereum Stack Exchange. <https://ethereum.stackexchange.com/>.
- [24] Gopal Gurram. 2021. Can we deploy same ERC20-token on different blockchains? <https://stackoverflow.com/questions/68802705/can-we-deploy-same-erc20-token-on-different-blockchains/68805158#68805158>.
- [25] Qing Huang, Dianshu Liao, Zhenchang Xing, Zhengkang Zuo, Changjing Wang, and Xin Xia. 2023. Semantic-Enriched Code Knowledge Graph to Reveal Unknowns in Smart Contract Code Reuse. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 147 (Sept. 2023), 37 pages. <https://doi.org/10.1145/3597206>
- [26] Inspex. 2024. How 20 Million \$OP Was Stolen from the Multisig Wallet (Not Yet) Owned by Wintermute. <https://inspexco.medium.com/how-20-million-op-was-stolen-from-the-multisig-wallet-not-yet-owned-by-wintermute-3f6c75db740a>.
- [27] Ruizhe Jia and Steven Yin. 2022. To EVM or Not to EVM: Blockchain Compatibility and Network Effects. In *Proceedings of the 2022 ACM CCS Workshop on Decentralized Finance and Security* (Los Angeles, CA, USA) (DeFi'22). Association for Computing Machinery, New York, NY, USA, 23–29. <https://doi.org/10.1145/3560832.3563442>
- [28] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) (CCS '16). Association for Computing Machinery, New York, NY, USA, 254–269. <https://doi.org/10.1145/2976749.2978309>
- [29] Fuchen Ma, Zhenyang Xu, Meng Ren, Zijing Yin, Yuanliang Chen, Lei Qiao, Bin Gu, Huizhong Li, Yu Jiang, and Jiaguang Sun. 2022. Pluto: Exposing Vulnerabilities in Inter-Contract Scenarios. *IEEE Transactions on Software Engineering* 48, 11 (2022), 4380–4396. <https://doi.org/10.1109/TSE.2021.3117966>
- [30] Martin Lundfall (@Mrchico). 2020. ERC-2612: Permit Extension for EIP-20 Signed Approvals. <https://eips.ethereum.org/EIPS/eip-2612>.
- [31] Ather Nawaz. 2012. A Comparison of Card-sorting Analysis Methods. In *APCHI '12. Proceedings of the 10th Asia Pacific Conference on Computer-Human Interaction*, Vol. 2. Association for Computing Machinery, United States, 583–592. <http://apchi2012.org/> The 10th Asia Pacific Conference on Computer Human Interaction. 2012 ; Conference date: 28-08-2012 Through 31-08-2012.
- [32] OpenZeppelin. 2024. openzeppelin-contracts. <https://github.com/OpenZeppelin/openzeppelin-contracts>.
- [33] Orderly Network. 2023. What is an EVM Compatible Chain? <https://medium.com/@orderlynetwork/what-is-an-evm-compatible-chain-46a7825adc4d>.
- [34] Giuseppe Antonio Pierro and Roberto Tonelli. 2021. Analysis of Source Code Duplication in Ethereum Smart Contracts. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 701–707. <https://doi.org/10.1109/SANER50967.2021.00089>
- [35] SlowMist. 2024. SlowMist - Focusing on Blockchain Ecosystem Security. <https://www.slowmist.com/>.
- [36] Soliditydeveloper. 2022. What is ecrecover in Solidity? <https://soliditydeveloper.com/ecrecover>.
- [37] Stackoverflow. 2024. Stack Overflow - Where Developers Learn, Share, & Build Careers. <https://stackoverflow.com/>.
- [38] Kairan Sun, Zhengzi Xu, Chengwei Liu, Kaixuan Li, and Yang Liu. 2023. Demystifying the Composition and Code Reuse in Solidity Smart Contracts. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) (ESEC/FSE 2023). Association for Computing Machinery, New York, NY, USA, 796–807. <https://doi.org/10.1145/3611643.3616270>
- [39] Tanish Gupta. 2023. EOAs vs Contracts: Understanding the Two Types of Ethereum Accounts. https://medium.com/@spacefactor1@m{tanish_gupta}/eoas-vs-contracts-understanding-the-two-types-of-ethereum-accounts-378f9402d0e8.
- [40] Trail of Bits. 2024. Trail of Bits. <https://www.trailofbits.com/>.
- [41] TRON. 2024. Decentralize The Web. <https://tron.network/>.
- [42] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and Why Your Code Starts to Smell Bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 403–414. <https://doi.org/10.1109/ICSE.2015.59>
- [43] Uniswap. 2021. Router02. <https://docs.uniswap.org/contracts/v2/reference/smart-contracts/router-02>.
- [44] Vechain. 2023. Contract address prediction. <https://docs.vechain.org/core-concepts/evm-compatibility/test-coverage/contract-address-prediction>.
- [45] Shuai Wang, Yong Yuan, Xiao Wang, Juanjuan Li, Rui Qin, and Fei-Yue Wang. 2018. An Overview of Smart Contract: Architecture, Applications, and Future Trends. In *2018 IEEE Intelligent Vehicles Symposium (IV)*. 108–113. <https://doi.org/10.1109/IVS.2018.8500488>
- [46] Zexu Wang, Jiachi Chen, Yanlin Wang, Yu Zhang, Weizhe Zhang, and Zibin Zheng. 2024. Efficiently Detecting Reentrancy Vulnerabilities in Complex Smart Contracts. *Proc. ACM Softw. Eng.* 1, FSE, Article 8 (jul 2024), 21 pages. <https://doi.org/10.1145/3643734>
- [47] Zexu Wang, Jiachi Chen, Peilin Zheng, Yu Zhang, Weizhe Zhang, and Zibin Zheng. 2024. Unity is Strength: Enhancing Precision in Reentrancy Vulnerability Detection of Smart Contract Analysis Tools. *IEEE Transactions on Software*

- Engineering* (2024), 1–12. <https://doi.org/10.1109/TSE.2024.3427321>
- [48] Wikipedia contributors. 2024. Binance — Wikipedia, The Free Encyclopedia. <https://en.wikipedia.org/w/index.php?title=Binance&oldid=1214652916>.
- [49] Wikipedia contributors. 2024. Ethereum — Wikipedia, The Free Encyclopedia. <https://en.wikipedia.org/w/index.php?title=Ethereum&oldid=1214478940>.
- [50] Wikipedia contributors. 2024. Polygon (blockchain) — Wikipedia, The Free Encyclopedia. [https://en.wikipedia.org/w/index.php?title=Polygon_\(blockchain\)&oldid=1214411541](https://en.wikipedia.org/w/index.php?title=Polygon_(blockchain)&oldid=1214411541).
- [51] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.
- [52] Jed R Wood and Larry E Wood. 2008. Card sorting: current practices and beyond. *Journal of Usability Studies* 4, 1 (2008), 1–6.
- [53] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. 2021. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts (*ASE '20*). Association for Computing Machinery, New York, NY, USA, 1029–1040. <https://doi.org/10.1145/3324884.3416553>
- [54] Shuo Yang, Jiachi Chen, and Zibin Zheng. 2023. Definition and Detection of Defects in NFT Smart Contracts (*ISSTA 2023*). Association for Computing Machinery, New York, NY, USA, 373–384. <https://doi.org/10.1145/3597926.3598063>
- [55] Jiashuo Zhang, Jiachi Chen, Zhiyuan Wan, Ting Chen, Jianbo Gao, and Zhong Chen. 2024. When Contracts Meets Crypto: Exploring Developers’ Struggles with Ethereum Cryptographic APIs. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) (*ICSE '24*). Association for Computing Machinery, New York, NY, USA, Article 164, 13 pages. <https://doi.org/10.1145/3597503.3639131>
- [56] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2021. Smart Contract Development: Challenges and Opportunities. *IEEE Transactions on Software Engineering* 47, 10 (2021), 2084–2106. <https://doi.org/10.1109/TSE.2019.2942301>

Received 2025-02-24; accepted 2025-03-31