EquivSage: LLM-assisted Detection of EVM-Inequivalent Code Smells in Multi-chain Reuse Contracts

ZEXU WANG, School of Cyberspace Security (School of Cryptography), Hainan University, China and Sun Yat-sen University, China
JIACHI CHEN, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, China
YANLIN WANG, Sun Yat-sen University, China
KAIWEN NING, Sun Yat-sen University, China and Peng Cheng Laboratory, China
ZEWEI LIN, Shantou University, China and Sun Yat-sen University, China
JINGWEN ZHANG, Sun Yat-sen University, China and Peng Cheng Laboratory, China
DAOYUAN WU, Lingnan University, China
ZIBIN ZHENG*, Sun Yat-sen University, China and Guangdong Engineering Technology Research Center of Blockchain, China

With the increasing 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 an empirical study to reveal the causes and characteristics of *EVM-Inequivalent Code Smells*. To ensure the identified smells reflect real developer concerns, we analyzed 1,379 security audit reports, 823 bug bounty 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 nine types of *EVM-Inequivalent Code Smells*. To enable efficient detection, we developed *EquivSage*, a tool that leverages the contextual understanding capabilities of *large language models* (LLMs) to guide slicing and static taint analysis via task-specific prompts. Symbolic execution is integrated to improve detection reliability. An analysis of 1,263,683 contracts across six EVM-compatible blockchains using *EquivSage*

*Corresponding Author

This research is supported by the National Natural Science Foundation of China (No. 62302534, 62332004), the Technology Program of Guangzhou, China (202103050004).

Authors' Contact Information: Zexu Wang, School of Cyberspace Security (School of Cryptography), Hainan University, Haikou, China and Sun Yat-sen University, Zhuhai, China, wangzx01@hainanu.edu.cn; Jiachi Chen, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou, China, chenjiachi@zju.edu.cn; Yanlin Wang, Sun Yat-sen University, Zhuhai, China, wangylin36@mail.sysu.edu.cn; Kaiwen Ning, Sun Yat-sen University, Zhuhai, China and Peng Cheng Laboratory, Shenzhen, China, ningkw@mail2.sysu.edu.cn; Zewei Lin, Shantou University, Shantou, China and Sun Yat-sen University, Zhuhai, China, linzw3@mail2.sysu.edu.cn; Jingwen Zhang, Sun Yat-sen University, Zhuhai, China and Peng Cheng Laboratory, Shenzhen, China, zhangjw273@mail2.sysu.edu.cn; Daoyuan Wu, Lingnan University, Hong Kong, China, daoyuanwu@ln.edu.hk; Zibin Zheng, Sun Yat-sen University, Zhuhai, China and Guangdong Engineering Technology Research Center of Blockchain, Zhuhai, China, zhzibin@mail.sysu.edu.cn.

ACM 1557-7392/2026/8-ART
https://doi.org/10.1145/3841635

reveals that, on average, 15.45% contain at least one *EVM-Inequivalent Code Smell*, underscoring its widespread prevalence. Since 2024, code smells in reused contracts on *Ethereum*, *Polygon*, and *Optimism* have increased significantly. While not all instances lead to financial loss, high frequency and asset exposure highlight the risks inherent in contract reuse. Developers are encouraged to avoid *copy-and-paste* practices and to detect such smells proactively before reuse.

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

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

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) [69, 83]. Currently, *Ethereum* has deployed over 82 million contracts and provided a comprehensive ecosystem for developers. To enable better reusability of contracts on *Ethereum*, more than 60% blockchains support the execution of EVM, such as *Binance Smart Chain* (BSC) [74], *Polygon* [76], and *Arbitrum* [9], which have already captured a significant market share [45, 52]. Meanwhile, an increasing number of blockchains originally designed with architectures independent of the EVM, such as *Solana* [44] and *Polkadot* [54], are adopting EVM compatibility. This development not only enables them to leverage the mature ecosystem of the EVM but also substantially reduces the time and effort required for developers to deploy applications across multiple chains.

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 on various EVM-compatible blockchains. For example, `Chain ID` [77] 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 [43]. The lack of validation of `Chain ID` in *Wintermute*'s *Ethereum* contract allowed attackers to exploit the reused contract on the *Optimism* blockchain through signature replay attacks, eventually resulting in huge financial loss.

Thus, it is important to ensure reusability and reliability when reusing smart contracts on other EVM-compatible blockchains. In software engineering, *code smells* are characteristics of the source code that can indicate deeper issues, representing design flaws in the program [65]. These flaws can affect the reusability, understandability, and reliability of the code. 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 inconsistencies caused by reused *Solidity* contracts across EVM-compatible blockchains, termed as *EVM-Inequivalent Code Smells*. To ensure that the identified smells reflect real developer issues and guide the reuse of secure contracts, we conducted an empirical study to uncover the causes and characteristics of *EVM-Inequivalent Code Smells*. Specifically, we analyzed security reports and *Stack Overflow* posts to collect security issues related to reused contracts on EVM-compatible blockchains, such as *Binance Smart Chain* (BSC) [74] and *Polygon* [76].

Through extensive data collection, we analyzed 1,379 security audit reports, 823 bug bounty reports, and 326 *Stack Overflow* posts. Using the *Open Card Sorting* method [78], we identified nine 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), *Gas Limit Imbalance* (GLI), *Broken Msg.sender == Tx.origin Assumption* (BMTA), *Inconsistent Opcodes Implementation* (IOI) and *Inconsistent Precompile Implementation* (IPI) (details in Section 3.3). These code smells hinder the reusability and introduce security risks for multi-chain reuse contracts. **Compared with our previous work, this study incorporates an additional 823 bug bounty reports. These reports, sourced from well-established platforms including Code4Rena [1], Immunefi [3], Sherlock [4], and CodeHawks [2], have been collaboratively reviewed and validated by**

developers with extensive security auditing experience, providing valuable complementary insights to traditional audit reports. Based on this expanded dataset, we introduce three new code smell types: BMTA, IOI, and IPI. Each of these arises from inconsistencies in *Ethereum Improvement Proposal* (EIP) upgrade support across EVM-compatible blockchains.

To investigate the prevalence of the defined *EVM-Inequivalent Code Smells*, we developed *EquivSage*, a detection tool that leverages contextual semantic understanding capabilities of *large language models* (LLMs) to improve program analysis for efficient and accurate detection. *EquivSage* first constructs the *Inter-contract Program Dependency Graph* (I-PDG) [71] to capture cross-contract data and control dependencies, providing semantic context for subsequent analyses. Leveraging *Tier of Thought* (ToT) [70], it designs task-specific prompts that guide the LLM to perform program slicing for extracting relevant code fragments and to identify sanitized variables for assisting static taint analysis. Finally, symbolic execution is applied to verify the feasibility of suspicious execution paths, further improving detection precision. **Compared with our previous tool, *EquivGuard* [72], which relied on fixed patterns, *EquivSage* combines expert-defined domain patterns with LLM-based contextual analysis and program-analysis verification. The domain patterns encode the knowledge required to identify critical operations, risk conditions, and sanitization criteria for each code smell. The LLM is then used to interpret these patterns within dependency-preserved sliced code, helping identify task-related variables, contextual dependencies, and semantically equivalent sanitization logic that may not be captured by fixed syntactic matching alone. The integration of symbolic execution further improves detection reliability by validating the feasibility of suspicious paths.**

As EVM compatibility has become widely adopted across multiple blockchains, we applied *EquivSage* to a dataset of 1,263,683 contracts collected from six mainstream EVM-compatible blockchains: *Ethereum*, *Binance Smart Chain* (BSC), *Arbitrum*, *Polygon*, *Optimism*, and *Avalanche*. **Compared with our prior work, which analyzed 905,948 contracts deployed on these blockchains up to January 2024, we extended the dataset by incorporating an additional 357,735 contracts deployed through July 2025.** Our analysis demonstrates the prevalence of *EVM-Inequivalent Code Smells*, with an average of 15.45% of contracts containing at least one instance. Among these, *Phishing Contract Address* (PCA), *Time Discrepancy Trap* (TDT), and *Fixed Gas Reentrancy* (FGR) are the most frequently observed in multiple chains. *EquivSage* achieves 97.70% precision and 98.77% recall by integrating *Large Language Model* assistance, which enhances path exploration during static taint analysis, and *Symbolic Execution* verification, which validates the reachability of the path. Further analysis reveals that these smells are predominantly concentrated in reused contracts on *Ethereum*, *Polygon*, and *Optimism*, with a sharp increase since 2024. These findings indicate that multi-chain reuse accelerates the cross-platform propagation of *EVM-Inequivalent Code Smells*, underscoring the need for greater developer awareness and proactive mitigation in multi-chain development.

The main contributions of this work are as follows:

- We present the most comprehensive study on *EVM-Inequivalent Code Smells*. By manually analyzing real-world security audit reports, bug bounty reports, and *Stack Overflow* posts, we identify and define nine different types of code smell, providing clear definitions and illustrative examples for each.
- We design *EquivSage*, a detection tool that leverages the contextual semantic understanding capabilities of LLMs to enhance program analysis for efficient and accurate detection. *EquivSage* achieves 97.70% precision and 98.77% recall by integrating *Large Language Model* assistance for more effective path exploration during static taint analysis, and *Symbolic Execution* validates path reachability.
- We present the largest and most up-to-date dataset of 1,263,683 real-world contracts, revealing that 15.45% contain at least one *EVM-Inequivalent Code Smells*. This work highlights the need for developers to identify such smells and conduct thorough multi-chain testing to mitigate inconsistencies introduced by contract reuse.

- We make the source code of *EquivSage*, together with all analysis results and the complete dataset, publicly available at https://zenodo.org/records/20512279.

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 [62]. 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 [77]. 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 0x818ec0a7 [1] differs between *Ethereum* and *Moonbeam*.
■ **EIPs.** *Ethereum Improvement Proposals* (EIPs), like *EIP-2612* which introduced the *permit()* function for offline signature validation [48], 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 [37]. This standard continuously evolves; for instance, EIP-1380 [8] reduced the gas cost for the call to self from 700 to 40. Additionally, many EVM-compatible blockchains have their own gas cost standards and refund mechanisms to offer lower transaction fees, such as BSC's BEP [14].
■ **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 [28].
■ **Hard Fork.** Hard Fork significantly changes blockchain protocols, resulting in a split that creates a new blockchain [17]. This process is often used to implement major changes that lead to two versions of the blockchain with distinct paths. This split allows the new protocol to integrate advanced features or security improvements, although 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 [19].

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 [52]. Table 1 presents the statistics of the 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 $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.

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

[1] 0x818ec0a7fe18ff94269904fced6ae3dae6d6dc0b

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%

7.5s, and the average transaction fee on *Polygon* is minimal, at $0.013. Although 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 to these blockchains, affecting EVM compatibility.

Table 2. Statistics of Top 5 EVM-Compatible Blockchains

Blockchain	Txn Fee	Block Time	TTF	EIP-compliant
Ethereum	$15.59	12.14s	16m	Fully
BSC	$0.24	3.01s	7.5s	Partial
Polygon	$0.013	2.26s	4m16s	Fully
Arbitrum	$0.28	0.26s	16m	Partial
Optimism	$0.069	2s	16m	Partial
Avalanche	$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 of real-world reports (including security audit and bug bounty reports) and *Stack Overflow* posts on EVM-compatible blockchains to define and classify *EVM-Inequivalent Code Smells*.

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. In addition, we collected 823 reports from well-known contract security bug bounty platforms, including *Code4Rena* [1], *Immunefi* [3], *Sherlock* [4], and *Codehawks* [2]. We then performed a thorough manual screening to extract 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* [36] and identified 30 teams that had open-sourced their audit reports, including *SlowMist* [58], *ConsenSys* [29], *BlockSec* [13], and *Trail of Bits* [63]. 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 the public websites of audit reports [32], we collected a total of 1,379 EVM-related audit reports.

3.1.2 Stack Overflow Posts Collection. *Stack Overflow* [60] is a popular Q&A community for developers, engineers, and technology enthusiasts. The platform uses a tag system to manage the 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 Contract Bug Bounty Reports Collection. The *Bug Bounty Report (BBR)* documents security vulnerabilities identified by bounty participants, along with detailed reproduction steps, exploit conditions, and remediation suggestions. These reports are jointly reviewed by multiple professional security auditors, who validate the identified issues, determine their severity levels, and provide practical insights into real-world vulnerability discovery. To collect bounty reports related to *EVM-Inequivalent Code Smells*, we compiled an initial dataset from leading smart contract security bounty platforms, including *Code4Rena* [1], *Immunefi* [3], *Sherlock* [4], and *Codehawks* [2], selected for their open-source availability and frequent citation in related work. These audit reports, each describing a single confirmed vulnerability, are published on GitHub. By filtering for confirmed and rewarded reports and searching with the keyword "*EVM*" and "*EVM Compatible*", we identified 823 reports.

3.1.4 Manual Screening. We employed source-specific retrieval strategies to accommodate the heterogeneity of terminology, document structure, and noise level across different data sources. For *Stack Overflow Posts*, where developers often describe EVM compatibility issues in informal, implicit, or problem-specific language, we used relatively broad tags and keywords (e.g., "*EVM*" and "*EVM Equivalent*") to increase recall. In contrast, bug bounty reports typically follow more standardized vulnerability-reporting conventions and often include extensive protocol-specific details that are orthogonal to EVM inequivalence. We therefore used more targeted keywords for these reports (e.g., "*EVM*" and "*EVM Compatible*") to improve precision and reduce irrelevant records. This source-specific retrieval process yielded 1,379 *SARs*, 326 *SOPs*, and 823 *BBRs* as candidate records. Importantly, these retrieval criteria were used only to construct the candidate pool; final inclusion decisions were made exclusively through the unified manual screening procedure.

To identify records relevant to *EVM-Inequivalent Code Smells*, two authors served as annotators and manually screened all candidate reports and posts. Both annotators are Ph.D. students whose research focuses on smart contract security. Each has published at least three papers on smart contract security in top-tier software engineering or security venues, and both have practical experience in smart contract vulnerability analysis and security auditing. This expertise provided the domain knowledge required to assess candidate records in terms of smart contract semantics, vulnerability patterns, and security-relevant execution behaviors. Since the retrieval process was intentionally designed to include potentially relevant records, the candidate pool inevitably contained issues unrelated to EVM inequivalence. The annotators therefore removed irrelevant records by examining the fields most informative for each source type. For *SOPs*, they reviewed the *Title*, *Description*, and *Comment*, which capture the reported problem and the subsequent discussion. For *SARs* and *BBRs*, they inspected the *Issue*, *Root Cause*, and *Recommendation* sections, which describe the vulnerability, its underlying cause, and suggested mitigation. Based on this evidence, the annotators determined whether each record involved EVM-Inequivalent behavior.

Two authors independently reviewed the reports and resolved discrepancies through discussion to reach consensus. Items that could not be immediately classified were marked as *tentative*. After comparing results, discussing disagreements and examining the *tentative* group, the authors finalized the relevant collection. For example, one report stated, "*The check on the given block numbers is incorrect*" [41]. The authors disagreed on its inclusion: one favored retention, while the other recommended removal. Further discussion showed that the issue arose from incorrect checks on `block.number`, which could lead to logical errors. However, since this

was unrelated to EVM inequivalence caused by contract reuse, the report was excluded. After reconciling all differences, the authors identified 336 relevant source records of contract reuse on EVM-compatible blockchains, from which 340 code-smell instances were extracted. To ensure transparency and auditability, the released artifact includes supplementary process materials documenting our screening and decision procedures.

3.2 Data Analysis

To classify *EVM-Inequivalent Code Smells*, we applied the *Open Card Sorting* method, a common approach to problem discovery and definition in software engineering [49]. For each report or post, we created a card with structured sections that enables readers to sort and filter the content according to their understanding and preferences. Because the formats of *SOP*, *SAR*, and *BBR* differ, we designed two card types tailored to their characteristics: the *SOP Card* and the *Report Card*. *SAR* and *BBR* share a similar structure, so both were standardized using the *Report Card* template.

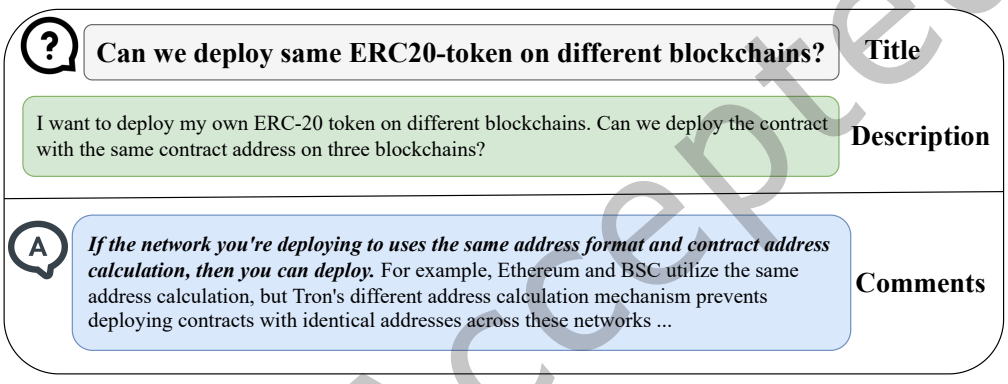


Fig. 1. An Example of the *SOP Card*

Figure 1 shows an example of the *SOP Card*, divided into three parts: *Title*, *Description*, and *Comments*. The author of the post wants to deploy ERC-20 tokens using the same contract address on multiple blockchains and seeks a solution. The responses in the *Comments* mention the possibility of deploying the contract with the same address format and deployment method on *Ethereum* [75] and *Binance Smart Chain* (BSC) [74], but the *Tron* network [64] 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 [39, 68]. 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).

Figure 2 shows an example of the *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 [48], and there was an additional `nonce` parameter in the input parameter. As `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 incorrect signature verification, which could lead to cross-chain replay attacks, particularly when reusing contracts across different EVM-compatible blockchains.

The same two annotators described in Section 3.1 classified code smells without predefined categories.

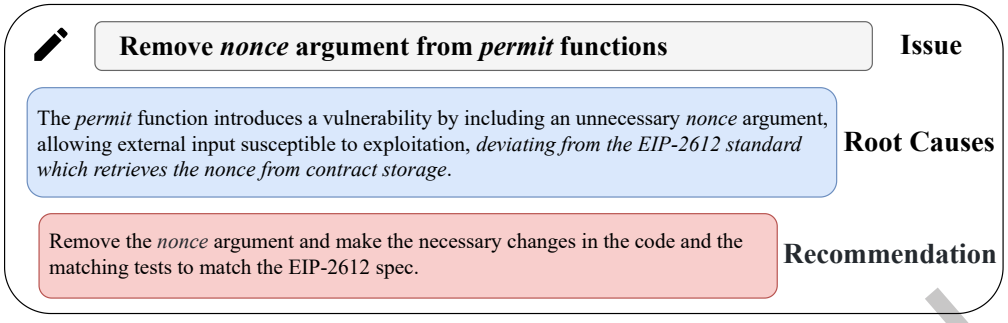


Fig. 2. An Example of the *Report Card*

❶ First Round. The researchers randomly selected 40% of the cards for initial classification. Each card was examined by reading the *Title (Issue)* and *Description* to understand the vulnerability, analyzing the code to identify the *Root Cause*, and reviewing the *Recommendations* to infer the related type. If a vulnerability did not fit any existing category, its representativeness and recurrence were evaluated before proposing a new category. Building on our previous work, six types of *EVM-Inequivalent Code Smells* had already been defined: *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). In this round, two additional types were identified: *Inconsistent Opcodes Implementation* (IOI) and *Inconsistent Precompile Implementation* (IPI).

❷ Second Round. The researchers independently classified the remaining 60% of the cards using the same procedure. In this process, they identified a new category, *Broken msg.sender == tx.origin Assumption* (BMTA), which had not been observed in the first round.

❸ Reconciliation. The researchers then compared their classification results. The main disagreements concerned the *Broken msg.sender == tx.origin Assumption* (BMTA) type. After discussion, they agreed that although this issue arises from account abstraction upgrades such as EIP-7702, the update fundamentally breaks the long-standing security assumption that `msg.sender == tx.origin` can reliably distinguish EOAs from contracts [15, 56]. This change transforms a previously accepted security pattern into a new systematic risk. After resolving all disagreements through discussion, the researchers reached consensus and confirmed nine types of *EVM-Inequivalent Code Smells*.

Table 3. Statistical Distribution of Different Code Smells Sources

Type	CCRA	TDT	PCA	GLI	FGR	BHM	BMTA	IOI	IPI
Quantity	90	45	28	64	23	24	17	25	24
SOP/SAR/BBR	0/82/8	15/17/13	11/0/17	0/38/26	0/13/10	0/21/3	0/6/11	0/8/17	0/4/20

Table 3 summarizes the distribution of nine types of code smells, showing their sources across *SOPs*, *SARs*, and *BBRs*. The concentration of cases in *SARs* and *BBRs* indicates that these issues are mainly discovered and addressed by security professionals. For example, *Cross-Chain Replay Attacks* (CCRA) are the most frequent, with 82 cases in *SARs* and 8 in *BBRs*, representing the largest overall share. In contrast, *Phishing Contract Address* (PCA) does not appear in *SARs* but is reported in 11 *SOPs* and 17 *BBRs*, showing that it is a problem developers encounter in practice and report through bounty programs. *Time Discrepancy Trap* (TDT) appears in all three sources (15 *SOPs*, 17 *SARs*, and 13 *BBRs*), underscoring its relevance in both developer discussions and formal security assessments. Overall, most code smells are reported in *SARs* and *BBRs*, suggesting that they are more

often identified by professional auditors or bounty hunters, while discussions in the broader developer community provide an important complement. These results highlight differences in their origins, offer insight into the security issues developers face, and emphasize the need for preventive measures when reusing contract code.

3.3 *EVM-Inequivalent Code Smells* Definition

In this section, we provide definitions of nine 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*

Code Smell	ID	Definition
Cross-Chain Replay Attack	CCRA	Incorrect chain information validation leads to cross-chain transaction replay.
Time Discrepancy Trap	TDT	The changing block time causes unexpected state updates.
Fixed Gas Reentrancy	FGR	Reentrancy vulnerabilities caused by changes in gas cost.
Block Height Misalignment	BHM	Different block heights cause inconsistent execution.
Phishing Contract Address	PCA	Stealing permissions from a designated contract address and impersonating it.
Gas Limit Imbalance	GLI	The specified gas limit in the contract leads to denial of service.
Broken `msg.sender == tx.origin` *Assumption*	BMTA	Using `msg.sender==tx.origin` for EOA checks becomes unsafe.
Inconsistent Opcodes Implementation	IOI	Different opcode behaviors across EVM-compatible chains.
Inconsistent Precompile Implementation	IPI	Different precompile behaviors across EVM-compatible chains.

(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 [48]. 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.

```
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  function verifyEIP712(address target, bytes32 hashStruct, uint8 v, bytes32 r, bytes32 s)
       public view returns (bool) {
8      bytes32 hash = keccak256(abi.encodePacked("\x19\x01", DOMAIN_SEPARATOR, hashStruct));
9      address signer = ecrecover(hash, v, r, s);
10     return(signer == target);
11 }
```

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

Code Example: In Figure 3, `verifyEIP712()` function (line 7) validates the signature information `(v, r, s)` to confirm the authenticity of the signer. It utilizes the `ecrecover()` to recover the signer's address and checks if the address matches the target address [59]. 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.

(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 [10]. 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.

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

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

Code Example: In Figure 4, the `TimeLockedWithdraw()` function calculates time intervals using `BLOCKS_PER_WEEK`, reducing reliance on timestamps and improving cross-chain deployment. However, in EVM-compatible blockchains with faster block generation, the 43,200 block time may not accurately represent a week, affecting the accuracy of the calculation. 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 contracts, such as for *Uniswap Routers* [67], a common practice in certain industries. However, when reusing contracts across multiple chains, these hard-coded addresses can lead to *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 in the *Wintermute* attack, attackers exploited signature replay and nonce collision techniques to take over target addresses in *Optimism* [43], impersonating the victim and launching phishing attacks.

```
contract SwapToken {
   // Uniswap Router address on mainnet
   address RouterAddr = 0x7a250d56...59F2488D;
   RouterV2 uniswapRouter = RouterV2(RouterAddr);
   // Exchange ETH for Token in Uniswap
   function swapEthForTokens(...) external payable {
      uniswapRouter.swapExactETHForTokens(...);
    }
}
```

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

Code Example: Figure 5 shows a contract utilizing *Uniswap*'s Router contract address [67] 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 abuse of blockchain resources. Developers often set a fixed gas limit in contracts. However, setting an excessively high or low gas limit can lead to issues [5]. 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 on 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 the 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 [28]. 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 can 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 limit is a vulnerable pattern that cannot fundamentally eliminate the occurrence of Reentrancy vulnerabilities [28].

(6) *Block Height Misalignment* (BHM). Block height, which is 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 an independent block height progression. The specific heights referenced in the code may not match the target heights of the blockchain. This mismatch can lead to severe consequences, such as the contract logic failing to execute the intended operations at the desired block heights or causing unintended consequences due to differences in block height progression between blockchains [11]. Attackers can exploit this, potentially leading to fund loss, governance issues, or critical failures.

```
// Hard fork for DAO
uint constant DAO_FORK_BLOCK = 1760000;
function handleFork() public {
    if (block.number >= DAO_FORK_BLOCK) {
    ...
}
```

Fig. 8. The Example of *Block Height Misalignment* (BHM)

Code Example: The `handleFork()` function in Figure 8 is designed to handle the DAO fork in *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 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.

(7) *Broken* $Msg.sender == Tx.origin$ *Assumption* (BMTA). The statement `require(msg.sender == tx.origin)` is a common pattern in smart contract development. Its purpose is to check whether the transaction's original initiator (`tx.origin`, typically an EOA) and its immediate caller (the contract executing the code) are the same account. This check is often used to restrict sensitive operations to EOAs, thereby mitigating risks such as phishing, front-running, and flash loan attacks. However, the introduction of the *Abstract Account* concept in EIP-7702 challenges this assumption [56]. EIP-7702 allows an EOA to temporarily attach and execute code within a single transaction [15]. When this attached code invokes another contract, the execution context identifies the caller as the EOA that signed the transaction, as the logic originates from the attached code. As a result, both `tx.origin` and `msg.sender` resolve to the same EOA address. Consequently, the condition `require(msg.sender == tx.origin)` always evaluates to true, undermining the original assumption that it distinguishes EOAs from contract accounts. This behavior may cause reused contracts that rely on this pattern to exhibit inconsistent security properties across EVM blockchains with different levels of support for EIP-7702.

Code Example: As shown in Figure 9, auditors found that the *OptimismPortal.sol* contract in the *Optimism* project [26] uses the condition `msg.sender != tx.origin` to verify the account. The core issue is that the security assumption of using `tx.origin` to determine whether the caller is a contract account becomes invalid with the introduction of account abstraction. Under EIP-7702, the distinction between `tx.origin` and `msg.sender` becomes ambiguous, rendering this check ineffective. Contracts relying on this logic may encounter functional

```
...
    if (msg.sender != tx.origin) {
        from = AddressAliasHelper.applyL1ToL2Alias(msg.sender);
     }
 ...
```

Fig. 9. The Example of *Broken Msg.sender == Tx.origin Assumption* (BMTA)

failures or unintended behavior when deployed on chains with different levels of EIP-7702 support. Such reliance on a fragile assumption constitutes an *EVM-Inequivalent Code Smell*, as its security properties may vary across platforms, increasing the risk of inconsistent behavior and potential vulnerabilities in multi-chain contract reuse.

(8) *Inconsistent Opcodes Implementation* (IOI). Although many blockchains claim EVM compatibility, their implementations of opcodes often diverge from those of *Ethereum*. As a result, the same code may deploy and execute correctly on one chain but fail, trigger run-time errors, or behave unexpectedly on another. Our analysis identifies two main sources of this problem. The first is *inconsistent opcode behavior*, where the same opcode produces different results or changes in state across chains, as in the cases of `COINBASE` [50] and `ORIGIN` [34]. The second is the *inconsistent opcode set*, where some chains have not yet implemented opcodes introduced in recent *Ethereum* mainnet hard forks, such as the `BLOBHASH` opcode from *Dencun* [24, 55]. These inconsistencies undermine the reusability and behavioral consistency of contracts in multi-chain environments.

```
/// @notice Simulates the EVM `extcodehash` opcode, returning the bytecode hash per EIP
    -1052.
function getCodeHash(uint256 _input) external view override returns (bytes32) {
    address account = address(uint160(_input));
    // Original zkSync Era behavior:
    // Return the hash of an empty string for precompiled contract addresses.
    // Differs from Ethereum EVM spec: omits balance check when determining "empty account"
    // (EVM considers code, nonce, and balance; omission may cause inconsistent results)
    if (uint160(account) <= CURRENT_MAX_PRECOMPILE_ADDRESS) {
        return EMPTY_STRING_KECCAK;
    }
    // Revised behavior:
    if (uint160(account) <= CURRENT_MAX_PRECOMPILE_ADDRESS && account.balance != 0) {
        return EMPTY_STRING_KECCAK; // Precompile with non-zero balance
    } else if (uint160(account) <= CURRENT_MAX_PRECOMPILE_ADDRESS && account.balance == 0) {
        return bytes32(0); // Precompile with zero balance
    }
    ...
}
```

Fig. 10. The Example of *Inconsistent Opcodes Implementation* (IOI)

Code Example: This case originates from the `getCodeHash()` in the `AccountCodeStorage` system contract of *zkSync Era* [27], which emulates the `EXTCODEHASH` opcode. As shown in Figure 10, *zkSync Era* implementation diverges from *Ethereum*, with the key difference being the criteria used to identify an empty account. According to the EVM specification, this determination should check the account's code, nonce, and balance, whereas *zkSync Era* implementation omits the balance check. Contracts that rely on the precise behavior of `EXTCODEHASH` for critical logic, such as distinguishing between an EOA with a balance and a truly empty account, may behave unpredictably

when executed in *zkSync Era*. This omission results in inconsistent EXTCODEHASH execution semantics. Such deviations undermine the reliability and portability of contract reuse and introduce logic errors and security vulnerabilities.

(9) *Inconsistent Precompile Implementation* (IPI). To mitigate the performance limitations of the EVM, some EVM-compatible blockchains introduce custom precompiled contract logic to support the efficient execution of complex operations across different architectures. However, when these implementations do not strictly follow the core semantics of the EVM, the resulting precompiled contracts exhibit behavioral inequivalence with Ethereum. This divergence appears in two main forms. First, differences in execution semantics: for example, in *zkSync Era*, the custom ECRECOVER precompiled contract returns a fixed zero value instead of the recovered signer address when invoked via DELEGATECALL, causing signature verification logic to fail [25]. Second, chain-specific precompiled contracts: for example, *Optimism* adds precompiled contracts for the verification of cross-domain messages and the aliasing of addresses, which have no counterparts on Ethereum [66]. Such variations in precompiled contract implementations can introduce security risks in decentralized applications deployed across multiple chains and reduce the reusability of *Solidity* contracts.

```
function ecrecoverDelegatecall() public returns (bytes32) {
    // Encode signature parameters (hash, v, r, s) into memory
    bytes memory data = abi.encode(h, v, r, s);
    assembly {
        // Call the ECRECOVER precompile (0x01) via delegatecall
        // On standard EVM: returns recovered signer address
        // On zkSync Era: returns all-zero value (inconsistent behavior)
        pop(delegatecall( gas(),
                0x01,  // ECRECOVER precompile contract address
                add(data, 0x20), mload(data), 0, 0x20))
        // Return the result (here, will be zero on zkSync Era)
        return(0, 0x20)
    }
}
```

Fig. 11. The Example of *Inconsistent Precompile Implementation* (IPI)

Code Example: As shown in Figure 11, calling the ECRECOVER precompiled contract (address 0x01) via DELEGATECALL on zkSync Era produces behavior inconsistent with Ethereum. The ecrecoverDelegatecall() function encodes the signature parameters (h, v, r, s) into memory and invokes the ECRECOVER through DELEGATECALL. In the EVM, this call returns the recovered signer address. In contrast, on *zkSync Era*, it returns a zero value. This anomaly breaks the expected equivalence of ECRECOVER and can cause contracts that rely on signature verification to fail, potentially leading to rejected transactions, locked funds, and other security risks. The issue highlights how inconsistencies in precompiled contract semantics across EVM-compatible chains threaten both security and portability.

4 Methodology

4.1 Overview

Figure 12 illustrates the workflow of *EquivSage*, which takes *Smart Contract Source Code* as input and produces *Detection Results*, including the type, location, and reasoning generated by the LLM. The detection process consists of three main phases: *Phase 1: Slicing with LLM Analysis*, *Phase 2: LLM-assisted Taint Analysis*, and *Phase 3: Symbolic Execution Verification*. To analyze the propagation of both data flow and control flow across contracts,

EquivSage constructs the *Inter-contract Program Dependency Graph* (I-PDG) [71], which models data and control dependencies between contracts. This graph provides complete dependency information to support program slicing and taint analysis. In addition, symbolic execution is applied to the *Control Flow Graph* (CFG) [46] to verify the reachability of paths.

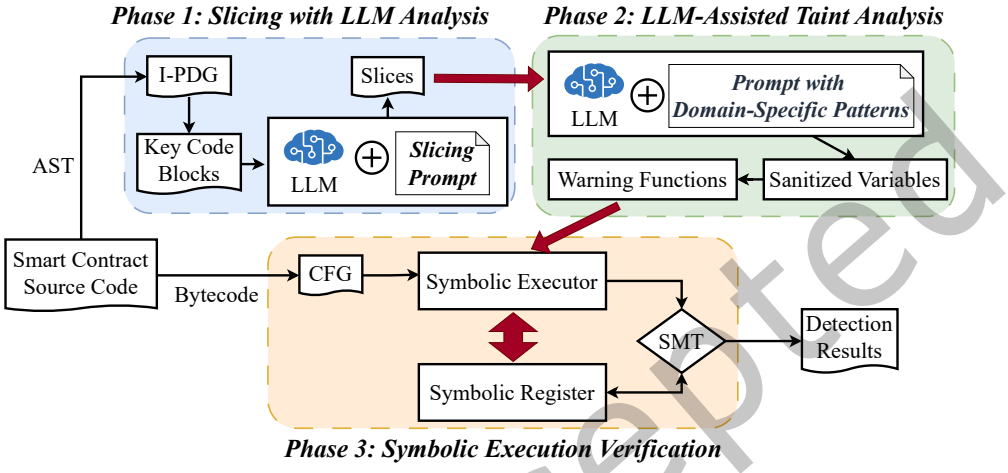


Fig. 12. The Workflow of *EquivSage*

Phase 1: Slicing with LLM Analysis. To extract relevant code segments and mitigate the input length limitations of LLMs, the *Slicing Prompt* is designed to guide LLM in the slicing of the program. Static analysis is first applied to quickly locate code blocks that contain key operations. These blocks are then provided to the LLM, which interprets their program context, identifies related functions and variables, and determines their relevance to the detection task. By leveraging the LLM's contextual understanding and the dependency relationships in the I-PDG, *EquivSage* refines the slices while preserving essential data and control flow information. The resulting slices include all functions required for execution, ensuring completeness for subsequent analysis.

Phase 2: LLM-assisted Taint Analysis. *EquivSage* augments static taint analysis with LLM-based contextual reasoning over the sliced code. This semantic enhancement is constrained by expert-defined domain knowledge rather than relying on unconstrained LLM inference. Specifically, domain-specific patterns encode the critical operations, risk conditions, and sanitization criteria associated with each smell type, and the LLM applies these patterns to dependency-preserved code slices. Guided by the *Prompt with Domain-Specific Patterns*, the LLM identifies sanitization logic that affects tainted variables within their surrounding program context and outputs the corresponding *Sanitized Variables*. These variables are then incorporated into the static taint analysis, enabling *EquivSage* to determine whether hazardous patterns remain along taint propagation paths and to report the functions containing such patterns as *Warning Functions*.

Phase 3: Symbolic Execution Verification. *EquivSage* employs symbolic execution to explore execution paths within *Warning Functions*, collect path constraints and verify their feasibility. This step is designed to avoid false positives caused by permission rules and improve the accuracy of detection. Finally, *EquivSage* compiles the type, location, and explanatory reasoning of each detected code smell into the *Detection Results*.

4.2 I-PDG Construction and Bytecode CFG Recovery

To support effective traversal of global execution paths, we integrate dependency analysis over *Abstract Syntax Tree* (AST) statement blocks with the *Inter-Contract Control Flow Graph* (I-CFG) [47], thereby constructing the *Inter-Contract Program Dependency Graph* (I-PDG) [71]. The resulting I-PDG serves as a structural abstraction for subsequent analyses, rather than as an independent mechanism for semantic classification. Specifically, the AST captures syntactic entities, such as variables, expressions, statements, and function calls, together with their hierarchical relations. The I-PDG enriches this syntactic representation by explicitly modeling control dependencies, data dependencies, and inter-procedural call/return relations. Consequently, the AST and I-PDG preserve the program context required for dependency-aware reasoning; however, this structural information alone is insufficient to determine whether an operation is security-critical or whether a code fragment constitutes an *EVM-Inequivalent Code Smell*.

```solidity
contract IpdgMini {
    address owner;
    uint256 chainId;
    bytes32 DOMAIN_SEPARATOR;
    mapping(address => bool) trustedRelayers;
    mapping(address => uint256) nonces;

    function batchTrust(address[] calldata rs) external {
        require(msg.sender == owner);
        for (uint256 i = 0; i < rs.length; i++) {
            if (rs[i] != address(0)) trustedRelayers[rs[i]] = true;
        }
    }

    function updateDomain(uint256 _chainId) external {
        require(msg.sender == owner || trustedRelayers[msg.sender]);
        chainId = (_chainId != block.chainid) ? _chainId : block.chainid;
        DOMAIN_SEPARATOR = keccak256(abi.encode(chainId, address(this)));
    }

    function buildDigest(address user, bytes32 h) internal view returns (bytes32) {
        return keccak256(abi.encodePacked(DOMAIN_SEPARATOR, h, nonces[user]));
    }

    function verifyPermit(address user, bytes32 h, uint8 v, bytes32 r, bytes32 s)
      external returns (bool)
    {
        address signer = ecrecover(buildDigest(user, h), v, r, s);
        if (signer == user) { nonces[user]++; return true; }
        return false;
    }
}
```

Fig. 13. A Compact Solidity Example for I-PDG Construction

Accordingly, *EquivSage* explicitly separates structural graph construction from semantic interpretation. The semantic layer is guided by expert-defined domain patterns that specify security-relevant operations and risk

conditions, and it is further instantiated through LLM-assisted contextual analysis over dependency-preserved code. For CCRA detection, `ecrecover()` is treated as a semantic anchor because it performs signature recovery, and cross-chain replay risks arise when the recovered signature is not bound to chain-specific information. Although the AST can locate the syntactic occurrence of `ecrecover()` and the I-PDG can characterize its dependencies with the digest, signature parameters, and relevant state variables, the security relevance of this invocation must be assessed by interpreting these dependencies in context. It cannot be reliably inferred from graph topology or rigid syntactic rules alone. This design enables the I-PDG to provide dependency-preserving program context, while expert-defined patterns and LLM-assisted semantic analysis jointly provide the basis for identifying *EVM-Inequivalent Code Smells*.

To illustrate the I-PDG construction procedure, we use the compact Solidity contract in Figure 13 as a representative example. Figure 14 shows the corresponding source-level I-CFG, where each numbered node is mapped to a statement-level program element in the example. Starting from the parsed AST, *EquivSage* derives the *Control Dependence Graph* (CDG) to capture dependencies between branch predicates and the statements whose execution they control. It also constructs the *Data Dependence Graph* (DDG) to model def-use relations induced by variable definitions, state-variable accesses, and value propagation. The I-PDG is then obtained by integrating control dependencies, data dependencies, and inter-procedural call/return relations into a unified source-level inter-contract representation. Consistent with the separation between structural modeling and semantic interpretation, *EquivSage* uses the I-PDG to recover dependency paths around security-relevant anchors. In the CCRA example, these paths allow static taint analysis to examine how the digest, signature parameters, nonce updates, and chain-related state are connected to the `ecrecover()` call through potentially risky dependencies.

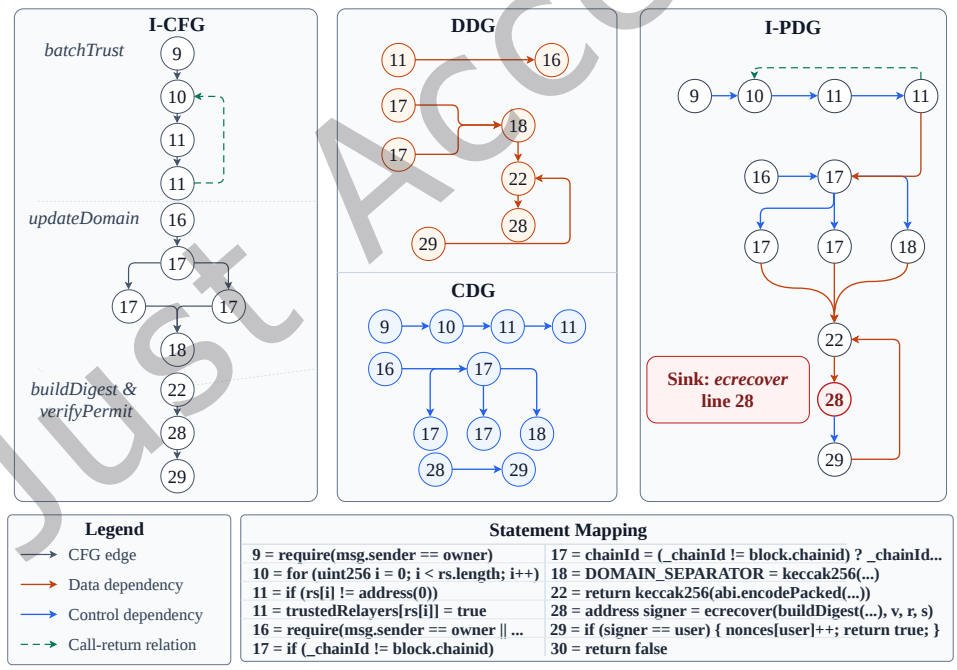


Fig. 14. I-PDG constructed from the contract in Figure 13, capturing loops, branches, internal calls, cross-function state updates, shared state-variable reads and writes, and the `ecrecover()` sink.

I-PDG Construction. Figure 13 presents the revised Solidity example, and Figure 14 shows the corresponding I-PDG constructed from this code. This example illustrates how the I-PDG captures source-level dependencies that span multiple functions, including control dependencies introduced by branch and loop predicates, data dependencies induced by state-variable reads and writes, and call/return dependencies between callers and callees. In particular, `batchTrust()` contains a loop and a branch condition that jointly control the update of `trustedRelayers`, which is later read by the permission check in `updateDomain()`, forming a cross-function data dependency visible as a data-dependency edge in Figure 14. Within `updateDomain()`, the predicate `_chainId != block.chainid` introduces a control-dependency edge that determines the value assigned to `chainId`, which further affects the update of `DOMAIN_SEPARATOR`. The function `buildDigest()` then reads `DOMAIN_SEPARATOR` and `nonces[user]` through data-dependency edges shown in Figure 14 to construct the digest passed by `verifyPermit()` to `ecrecover()` via a call/return edge. After successful verification, `verifyPermit()` updates `nonces[user]`, thereby influencing subsequent digest construction. These dependencies connect the permission condition, chain-specific state update, digest construction, signature recovery, and nonce update within a unified graph representation.

The example shows why the I-PDG is necessary for subsequent slicing and taint analysis. As illustrated in Figure 14, starting from the `ecrecover()` node, *EquivSage* can traverse data-dependency and call/return edges backward to retain the digest construction, the read of `DOMAIN_SEPARATOR`, the read and update of `nonces[user]`, and the prior update of `DOMAIN_SEPARATOR` in `updateDomain()`. As a result, *EquivSage* analyzes not only the local statements around `ecrecover()`, but also the cross-function context required to determine whether the signature is bound to chain-specific information and whether nonce-based replay protection is complete.

Algorithm 1 formalizes the I-PDG construction. The graph is constructed over statement-level program elements extracted from the AST and the source-level inter-contract CFG, rather than over source-code lines, entire functions, or complete contracts. Each node may correspond to a variable declaration, assignment, branch or loop predicate, function call, return statement, state-variable read, or state-variable write. For example, `if (_chainId != block.chainid)`, `DOMAIN_SEPARATOR = keccak256(...)`, `digest = buildDigest(...)`, and `signer = ecrecover(...)` are represented as separate nodes. Function entry and exit points are introduced as special nodes to model interprocedural call and return relationships. After adding source-level CFG, call/return, and control-dependency edges, *EquivSage* computes data-dependency edges using reaching-definition analysis. For each statement-level node n, $\mathsf{Use}(n, AST)$ extracts the variables read by n, whereas $\mathsf{Modify}(n, AST)$ extracts the variables written or updated by n; for instance, `nonces[user]++` yields the modified-variable set {`nonces[user]`}. For each variable $v \in \mathsf{Use}(n, AST)$, the algorithm adds a data-dependency edge from every reaching definition in $\mathsf{Def}[v]$ to n. When a mapping key or array index cannot be statically resolved, *EquivSage* adopts conservative abstractions, such as `nonces[*]` or `trustedRelayers[*]`. Thus, the construction operates over variable sets and does not require selecting a single variable from a successor node.

Bytecode CFG Recovery. Separately from the source-level I-PDG, *EquivSage* constructs a bytecode-level CFG for symbolic execution. This CFG is used to enumerate executable bytecode paths and collect path constraints, rather than to represent source-level semantic dependencies. To recover jump targets precisely, *EquivSage* converts bytecode stack operations into *Static Single Assignment* (SSA) form and applies constant propagation to resolve dynamically computed jump addresses. Following the SSA-based strategy in [71], the recovered bytecode CFG enables the symbolic executor to traverse feasible paths and validate suspicious executions.

4.3 Phase 1: Slicing with LLM Analysis

To manage the context-window limitations of LLMs while preserving analysis-relevant information [20, 82], *EquivSage* applies dependency-aware function-level slicing over the I-PDG prior to LLM invocation. Starting from the *Key Code Blocks* identified as slicing seeds, Phase 1 traverses data-dependency, control-dependency,

Algorithm 1: Constructing I-PDG

Input: Statement-level inter-contract CFG $\mathcal{G}_{cfg}$ and AST
Output: Inter-contract PDG $\mathcal{G}_{pdg}$

$\mathcal{G}_{pdg} \leftarrow \emptyset$; Def $\leftarrow \emptyset$;
foreach *statement-level node* $n \in \mathcal{G}_{cfg}$ **do**
  Add n and its CFG successors into $\mathcal{G}_{pdg}$;
  Add CFG edges (n, s) for each successor s of n;
end
foreach *call site* $c \in \mathcal{G}_{cfg}$ **do**
  foreach *callee* $f \in$ ResolveCallee(c) **do**
    Add call edge $(c, \text{entry}(f))$ and return edge $(\text{exit}(f), \text{succ}(c))$;
  end
end
foreach *branch or loop condition node* $b \in \mathcal{G}_{cfg}$ **do**
  Add control-dependency edges (b, n) for all $n \in$ CtrlDep(b);
end
repeat
  foreach *statement-level node* $n \in \mathcal{G}_{cfg}$ *in traversal order* **do**
    UseSet $\leftarrow$ Use(n, *AST*);
    ModSet $\leftarrow$ Modify(n, *AST*);
    foreach $v \in$ UseSet **do**
      Add data-dependency edges (d, n) for all $d \in$ Def$[v]$;
    end
    foreach $v \in$ ModSet **do**
      Def$[v] \leftarrow$ Def$[v] \cup \{n\}$;
    end
  end
until Def *reaches a fixed point*;
return $\mathcal{G}_{pdg}$;

and call/return edges to retain functions that directly or indirectly affect the seed operations. The resulting slice preserves cross-function control flow, state-variable reads and writes, and sanitization-related checks, while excluding unrelated business logic. The LLM then receives this dependency-preserved code context rather than isolated statements or the entire contract. This design reduces irrelevant input that may degrade LLM reasoning stability, while retaining the structural and semantic information required for subsequent task-related variable identification and sanitization analysis. Nevertheless, completeness of the sliced context is bounded by the precision of static dependency resolution; implicit dependencies or non-standard low-level constructs that fall outside the scope of AST-level pattern matching may result in an incomplete slice, which could in turn affect the quality of LLM-generated guidance.

The *Key Code Blocks* are determined by the characteristics of each code-smell type and correspond to security-critical operations or chain-dependent primitives. For example, `ecrecover()` is a key operation for detecting *Cross-Chain Replay Attack* (CCRA), whereas `gasleft()` is the key instruction for *Gas Limit Imbalance* (GLI). Detailed criteria for identifying *Key Code Blocks* are provided in Section 4.6, *EVM-Inequivalent Code Smells Detection*. Although *EquivSage* uses LLMs for semantic reasoning, these AST-level patterns are used only to localize candidate anchors before LLM invocation, not to make final detection decisions. This design is motivated

by the observation that *EVM-Inequivalent Code Smells* are often triggered by specific operations embedded in extensive business logic. For instance, TDT-related smells are commonly associated with `block.number`, CCRA-related smells are tied to signature recovery logic such as `ecrecover()`, and IPI-related smells involve invocations of precompiled contract addresses. Providing the entire contract to the LLM would introduce substantial irrelevant context, increase analysis cost, and reduce reasoning stability. Static localization therefore narrows the analysis scope, while the subsequent LLM-assisted slicing step interprets the sliced code in context. To address the limited semantic awareness of static localization, we design the *Slicing Prompt* to guide the LLM in identifying variables and logic relevant to the detection task based on contract semantics and intent. Specifically, when static analysis identifies *Key Code Blocks* associated with any of the nine *EVM-Inequivalent Code Smells*, *EquivSage* provides the LLM with their enclosing function names *%Warning Functions%* and the corresponding smell categories *%Warning_Code_Smell_Type%* as inputs for contextual slicing.

The *Slicing Prompt* is designed following the *Tier of Thought* (ToT) strategy [70] and established prompting practices [40]. It decomposes the slicing task into a structured reasoning procedure in which the output of each stage serves as the input to the subsequent stage, thereby improving the consistency of LLM-assisted analysis. To facilitate reliable downstream processing, the LLM is required to produce a JSON-formatted result only after completing the final stage. As shown in Figure 15, the prompt consists of four components: *Role Playing*, *Task Definition*, *Step-by-step Analysis*, and *Output Format*. The *Role Playing* component instructs the LLM to reason as a smart contract security auditor, while the *Task Definition* specifies that variables related to *%Warning_Code_Smell_Type%* should be extracted from *%Warning Functions%*. The *Step-by-step Analysis* decomposes the task into three decisions: (1) determine whether the execution of *%Warning Functions%* is related to *%Warning_Code_Smell_Type%*; (2) if the relation holds, extract all state variables in *%Warning Functions%*; and (3) identify which variables from step 2 influence the execution of *%Warning_Code_Smell_Type%*. The *Output Format* constrains the final response to a JSON object with predefined fields. By combining dependency-preserved slices with structured contextual reasoning, *EquivSage* reduces key-variable misidentification caused by insufficient semantic context and improves slicing precision.

Role Playing: Act as a smart contract security auditor proficient in vulnerability identification and mitigation, and complete the following task based on the provided instructions.

Task Definition: Extract variable names related to *%Warning_Code_Smell_Type%* from the *%Warning Functions%*.

Step-by-Step Analysis:

(1) Determine whether the execution of *%Warning Functions%* is related to *%Warning_Code_Smell_Type%*.
(2) If step 1 returns "Yes", extract all state variables from *%Warning Functions%*.
(3) From the variables identified in step 2, determine which ones influence the execution of *%Warning_Code_Smell_Type%* and output them.

Output Format: JSON object containing:

- `is_related_to_code_smell`: yes/no
- `all_involved_variables`: list of all variables involved
- `variables_related_to_code_smell`: list of variables determined

Fig. 15. *Slicing Prompt* Template Design

To balance semantic completeness with input-size constraints, *EquivSage* adopts a dependency-aware function-level slicing strategy. Using the task-related variables identified by the LLM and the global dependencies captured by the I-PDG, *EquivSage* retains functions whose statements have direct or indirect dependencies on the corresponding *Key Code Blocks*. Each function is treated as an analysis unit rather than being decomposed into isolated statements; once any statement in a function is dependency-relevant, the entire function is included in the slice. For example, in Figure 13, the `ecrecover()` call in `verifyPermit()` is a CCRA-related key operation. Its digest is produced by `buildDigest()`, which reads `DOMAIN_SEPARATOR` and `nonces[user]`, while `DOMAIN_SEPARATOR` is updated in `updateDomain()`. These functions are therefore retained together to preserve the dependency context needed for replay-risk analysis. This coarse-grained slicing strategy [12] reduces the risk of omitting state updates, control predicates, interprocedural calls, or sanitization logic required for contextual semantic interpretation. Meanwhile, the LLM receives function-level sliced code rather than the full contract, which suppresses irrelevant business logic and focuses the analysis on code regions associated with the target smell. To further mitigate context loss in complex implementations, such as customized low-level code or inline assembly, Phase 1 constructs slices around *Key Code Blocks*, LLM-identified task-related variables, and I-PDG-based dependency paths. The resulting LLM outputs are used only as intermediate semantic guidance; final detection is deferred to downstream static taint analysis and symbolic execution, which verify dependency propagation, path reachability, and path constraints before a report is produced.

4.4 Phase 2: LLM-assisted Taint Analysis

To enhance semantic understanding, *EquivSage* employs prompt guidance combined with domain-specific patterns to allow LLM to automatically identify variable sanitization operations and provide this information to static taint analysis for verification. In static taint analysis, sanitization removes sensitive data from tainted variables, allowing verification of whether security measures exist along the propagation path of taint [80]. This process supports the precise identification of actual vulnerabilities and reduces false positives for data paths that have already been securely handled. However, due to diverse vulnerability patterns and complex contextual semantics, many existing static taint analysis tools still rely on fixed rules or manual inspection to identify sanitization operations, which limits their effectiveness and efficiency. To address this, *EquivSage* leverages the LLM's ability to interpret the contextual semantics of contract code. The LLM determines whether sanitization operations are present and, if so, provides the related variables to static taint analysis for further verification, ensuring both rapid and accurate detection.

The *Prompt with Domain-Specific Patterns* translates the expert knowledge required for each *EVM-Inequivalent Code Smell* into a structured LLM-assisted sanitization-identification task. As shown in Figure 16, the prompt takes the sliced code from Phase 1, denoted by *%provided_code%*, and guides the LLM to determine whether variables associated with a given *%sanitized_type%* are properly constrained or sanitized. It consists of four sections. The *Role Playing* section assigns the LLM the role of a smart contract security auditor, while the *Task Definition* specifies the target sliced code and the sanitization category to be analyzed. The *Step-by-Step Analysis* section further decomposes the task into three stages: (1) *Search Variables*, which identifies state variables involved in the sliced code; (2) *Identify Key Variables*, which selects task-relevant variables according to *%sanitized_variable_identification_rules%*; and (3) *Verify Operation Process*, which checks whether these variables are sanitized through the methods specified in *%sanitization_methods%*. The *Output Format* section constrains the result to the required fields for downstream analysis. The reference table in the prompt provides smell-specific identification rules and sanitization methods for all nine smell types. In this table, *%sanitized_type%* denotes the smell category associated with the variable under inspection, *%sanitized_variable_identification_rules%* specify how variables requiring sanitization should be recognized, and *%sanitization_methods%* describe the operations or checks that can mitigate the corresponding risk. For example, CCRA analysis focuses on signature verification

Role Playing: Act as a smart contract security auditor proficient in vulnerability identification and mitigation, and complete the following task based on the provided instructions.

Task Definition: Analyze the provided code *%provided_code%* (generated from Phase 1 slicing) to determine whether it contains sanitized variables of category *%sanitized_type%*.

Step-by-Step Analysis:

(1) Identify all state variables involved during the execution of the sliced code.
(2) Determine the relevant state variables based on the provided *%sanitized_variable_identification_rules%*.
(3) Using the given *%sanitization_methods%*, determine which state variables related to *%sanitized_type%* are sanitized during function execution.

Output Format: JSON object containing:

- `is_related_to_code_smell`: yes/no
- `all_involved_variables`: list of all variables involved
- `variables_related_to_code_smell`: list of variables sanitized during execution

Reference Table: Identification Rules and Sanitization Methods

sanitized _type	**sanitized_variable _identification_rules**	**sanitization _methods**
CCRA	Use `block.chainId`	Verify chain ID in signature checks
TDT	Use `block.number` for time	Avoid fixed block time
FGR	Use `transfer()`/`send()` fixed gas	Prefer `call` and C-E-I pattern
BHM	Fixed block height constants	Do not key logic to one block
PCA	Hard-coded external addresses	Use params/immutables at deploy
GLI	Fixed gas thresholds (e.g., `gasleft() > X`)	Avoid hard gas gates; use fail-safe checks
BMTA	`msg.sender`/`tx.origin` EOA checks	Do not rely on `msg.sender == tx.origin`;
IOI	Depend on specific opcode behavior	Avoid relying on a single opcode's behavior
IPI	Depend on precompile return values	Avoid relying on a precompile's return value

Fig. 16. *Prompt with Domain-Specific Patterns* Template Design

logic and checks whether chain-specific information, such as `Chain ID` or blockchain identity, is bound to the signed message. Thus, the reference table provides explicit semantic guidance for interpreting sanitized variables while keeping the detection process grounded in expert-defined smell semantics. In this design, the LLM does not define the code-smell taxonomy or detection rules independently. Instead, it interprets the dependency-preserved sliced code and outputs sanitized-variable information as intermediate evidence for subsequent taint analysis and symbolic execution.

To prune irrelevant paths, *EquivSage* performs global data flow analysis using the I-PDG. The I-PDG models global function calls and cross-contract calls as code block jumps, ensuring correct handling of return values and enabling comprehensive global data flow analysis. *EquivSage* designates critical instructions as sinks and marks key variables influenced by external inputs as sources. Through backward taint analysis, it then traces the propagation of potentially unsafe data to locate security risks. Finally, sanitized-variable information provided by the LLM is incorporated to guide the analysis of sanitization operations and other security-sensitive behaviors. As shown in Figure 17, for the static taint analysis of *Cross-Chain Replay Attacks* (CCRA), *EquivSage* treats the

`ecrecover()` call as the *taint sink*. By applying backward dependency analysis on program dependencies, it identifies all variables that influence the *sink*, including assignments, parameter passing, and conditional branches, and checks whether they originate directly or indirectly from external input or user-controlled data. Such variables are marked as *taint sources*. In the code of Figure 3, line 9 is marked as the *sink* due to the `ecrecover()` call. The backward dependency analysis shows that `DOMAIN_SEPARATOR` in line 1 and `chainId` in line 2 are both tainted by user-defined external data (as indicated by the `setter()` function logic) and are therefore marked as *sources*. From the dependency relationships between these *sources* and the *sink*, *EquivSage* derives two distinct paths. It then incorporates the sanitized variables identified by the LLM to determine whether these paths include effective constraints or sanitization operations. If such measures are detected, the path is considered free of code smells; otherwise, it is flagged as containing a code smell. As illustrated in Figure 17, *EquivSage* starts from *taint sinks* along different paths and applies dependency analysis to determine whether constraints or dependencies exist on the *Blockchain ID* obtained from the blockchain environment. This reverse tracing enables the analysis of contamination relationships between *sources* and *sinks*, and whether the data have been validated, sanitized, or otherwise protected, thereby supporting efficient path property verification. For different *EVM-Inequivalent Code Smells*, different critical instructions are selected to guide efficient path exploration.

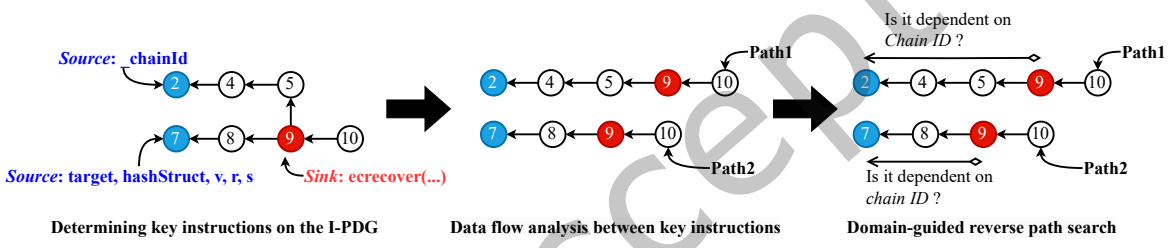


Fig. 17. Domain-Guided Static Taint Analysis for *Cross-Chain Replay Attack* (CCRA). The numbers in the circles are the line numbers from Figure 3. Blue circles are *Source* points, red circles are *Sink* points, and arrows show program dependencies.

4.5 Phase 3: Symbolic Execution Verification

To improve detection accuracy, *EquivSage* employs symbolic execution to traverse suspicious paths and verify their feasibility. To reduce false positives, symbolic execution is employed to collect path constraints and check the reachability of suspicious paths, thereby enhancing the precision of code smell detection. *EquivSage* traverses the CFG branches to model contract storage and collect path constraints, then proves reachability through constraint solving. To address the challenge of collecting incomplete path constraints, we introduce the *Symbolic Register*, which stores symbolic states and path constraint information. By analyzing bytecode storage operations and integrating the *Z3 solver*, specific values and symbolic expressions are stored as key–value pairs in the *Symbolic Register*.

As shown in Figure 18 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 the verification of attributes. 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,

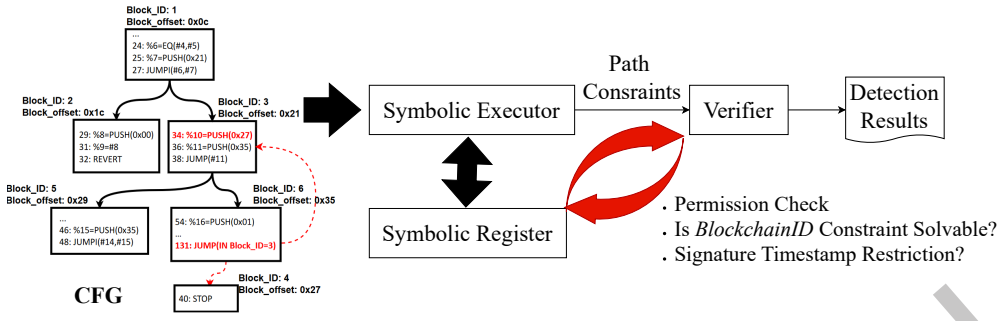


Fig. 18. Symbolic Execution Verification for *Cross-Chain Replay Attack* (CCRA)

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

During symbolic-execution-based verification, *EquivSage* invokes Z3 to determine the satisfiability of the collected path constraints. The solver returns one of three outcomes: SAT, UNSAT, or UNKNOWN. A SAT result establishes that the corresponding suspicious path is feasible, and the path is therefore retained as a confirmed report. In contrast, a UNSAT result proves that the path is infeasible, so the path is pruned. For an UNKNOWN result, *EquivSage* adopts a conservative treatment. Since UNKNOWN usually reflects the solver's inability to decide the constraints within its reasoning capability or resource budget, it does not provide sufficient evidence for either confirmation or refutation. Therefore, *EquivSage* neither reports such a path as a confirmed smell nor removes it as an infeasible path. Instead, the path is moved to an inconclusive set and recorded separately from the confirmed detection results. This design makes solver uncertainty explicit, prevents undecidable queries from inflating confirmed reports, and avoids unsoundly discarding potentially feasible paths. During ground-truth evaluation, if an UNKNOWN query is associated with a ground-truth positive sample, the sample is classified as a false negative only when no alternative SAT path exists for that sample. This conservative strategy may reduce recall but does not inflate precision. Consequently, only paths with solver-validated feasibility contribute to the final confirmed results, while UNKNOWN cases remain available for inspection and reproducibility.

4.6 *EVM-Inequivalent Code Smells* Detection

In this section, we present the criteria for selecting the *Key Code Blocks* of different types and the detection process.

(1) *Cross-Chain Replay Attack* (CCRA). *EquivSage* 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, *EquivSage* 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). *EquivSage* 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. *EquivSage* then determines if these variables are constants and examines the judgment conditions of the calculation results. Finally, symbolic execution verifies the feasibility of the path and potential inconsistencies across different EVM-compatible blockchains, ensuring accurate detection.

(3) *Phishing Contract Address* (PCA). *EquivSage* analyzes whether contracts contain unrestricted method calls to fixed contract addresses. To reduce false positives, *EquivSage* 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). *EquivSage* analyzes contracts to detect operations setting specific gas limits. Specifically, GLI queries the remaining gas amount using the `gasleft()` function [35] and adjusts the execution 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, the GLI is flagged.

(5) *Fixed Gas Reentrancy* (FGR). *EquivSage* 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) [2] [16, 73]. If the contract includes `transfer()/send()` methods but does not adhere to the safe *Check->Effect->Interaction* development pattern, then *EquivSage* considers this smell to exist.

(6) *Block Height Misalignment* (BHM). *EquivSage* 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. In addition, it performs a taint analysis to track the propagation of variables related to block heights and determine their impact on control flow decisions. Furthermore, *EquivSage* employs symbolic execution to verify the feasibility of identified paths that involve block height restrictions, ensuring accurate detection.

(7) *Broken msg.sender == tx.origin Assumption* (BMTA). *EquivSage* examines whether the contract includes conditional checks involving `msg.sender` and `tx.origin`. Specifically, it employs I-PDG analysis to identify conditional constraints related to these variables. Based on this, taint analysis is conducted to trace the propagation paths of `msg.sender` and `tx.origin` and to assess whether conditional constraints affect path execution. Finally, *EquivSage* applies symbolic execution to traverse the paths, thereby verifying the accuracy and reliability of the analysis results.

(8) *Inconsistent Opcodes Implementation* (IOI). Based on the *Total Value Locked* (TVL) ranking on *Defillama* [33], we examined the official documentation of the top ten EVM-compatible chains to evaluate their support for the 149 opcodes defined in the standard EVM. We then compiled a set of opcodes that differ from *Ethereum* in terms of availability (see Table 5). The results reveal inconsistencies in execution semantics (e.g., `COINBASE` and `EXTCODEHASH` return different values or behave differently on some chains), missing opcodes (such as `PUSH0`, `PREVRANDAO`, and `BLOBHASH`, which are absent on certain chains), and the removal or restricted use of legacy instructions (such as `CALLCODE` and `SELFDESTRUCT`). In the detection process, *EquivSage* locates these inconsistent opcodes through their corresponding *Solidity* representations, then applies taint analysis to trace invocation paths and identify critical business logic dependent on these instructions. Finally, symbolic execution is used to verify the reachability of the corresponding control-flow paths, enabling precise identification and confirmation of potential risks.

(9) *Inconsistent Precompile Implementation* (IPI). Based on the official documentation of the top ten EVM-compatible chains by TVL on *Defillama* [33], we compared their support for the eleven precompiled contracts defined in the *Ethereum* standard (see Table 6). For example, `RIPEMD-160` and `BLAKE2f` are not supported on *Scroll* and *zkSync Era*, `modexp` is missing on *zkSync Era*, `point evaluation` is available only on *Ethereum* and *Base*, while `secp256r1` is a chain-specific extension supported by *Polygon*, *Base*, and *zkSync Era*. These findings indicate that the inconsistencies mainly stem from differences in the implementation and support of *Ethereum*'s

[2] C-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.

Opcode	Arbitrum	Optimism	Polygon	Base	Linea	Scroll	ZKsync Era
`tx.origin`	✓	✗	✓	✗	✓	✓	✓
`blockhash(x)`	✗	✓	✗	✓	✓	✓	✓
`block.coinbase`	✗	✓	✓	✓	✓	✗	✗
`block.number`	✗	✓	✓	✓	✓	✓	✓
`block.prevrandao`	✗	✗	✓	✗	✗	✗	✗
`gaslimit`	✓	✓	✓	✓	✓	✓	✗
`blobhash(index)`	✓	✗	✗	✗	✗	✗	✗
`block.blobbasefee`	✗	✗	✗	✗	✗	✗	✗
`tload(key)`	✓	✓	✓	✓	✗	✓	✓
`tstore(key,value)`	✓	✓	✓	✓	✗	✓	✓
`mcopy(...)`	✓	✓	✓	✓	✗	✓	✓
`selfdestruct`	✓	✓	✗	✓	✓	✗	✗
`PUSH0`	✓	✓	✓	✓	✗	✓	✓
`callcode`	✓	✓	✓	✓	✓	✓	✗
`delegatecall`	✓	✓	✓	✓	✓	✓	✗
`msg.sender`	✗	✗	✓	✗	✓	✓	✓
Legend: ✓= Supported, ✗= Not Supported							

Table 5. Opcode support differences across top EVM-compatible chains

Precompile Contract (Address)	Ethereum	Polygon	Base	Linea	Scroll	ZKsync Era
RIPEMD-160 (0x...03)	✓	✓	✓	✓	✗	✗
modexp (0x...05)	✓	✓	✓	✓	✓	✗
blake2f (0x...09)	✓	✓	✓	✓	✗	✗
point evaluation (0x...0A)	✓	✗	✓	✗	✗	✗
secp256r1 (0x...0100)	✗	✓	✓	✗	✗	✓
Legend: ✓= Supported, ✗= Not Supported						

Table 6. Support for Ethereum precompiled contracts across top EVM-compatible chains

standard precompiled contracts. In the detection process, *EquivSage* first identifies the inconsistent precompile implementations listed in Table 6. It then inspects any calls to these precompile addresses via `CALL`, `STATICCALL`, or `DELEGATECALL` and analyzes the dependencies of the returned data. By capturing such invocation behaviors, *EquivSage* integrates I-PDG and taint propagation analysis to trace paths, revealing critical dependencies on

precompile return values within business logic. Finally, symbolic execution verifies the reachability of paths, enabling precise detection and confirmation of risks.

5 Evaluation

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

- RQ1: How does *EquivSage* perform on real large-scale datasets?
- RQ2: What is the performance of *EquivSage* in detecting *EVM-Inequivalent Code Smells*?
- RQ3: What is the impact of different phases within *EquivSage*?
- RQ4: What is the prevalence and distribution of *EVM-Inequivalent Code Smells* in multi-chain reused contracts?

5.1 Experimental Setup

Implementation: *EquivSage* is implemented in *Python*. It extends *Slither* [31] with enhanced data flow analysis to construct the I-PDG and perform dependency analysis, and extends *Rattle* [30] with the SSA-based approach to recover *control-flow graph* (CFG) paths for symbolic execution. The LLM component is used as semantic guidance within this program-analysis pipeline, rather than as a standalone vulnerability detector. Accordingly, *EquivSage* adopts *DeepSeek-V3.1* [7] as the default backend because the model selection prioritizes reliable code understanding, sufficient context capacity, stable responses, and controllable cost under repeated invocations. These requirements match its role in slicing guidance, task-related variable identification, and sanitized-variable recognition; *DeepSeek-V3.1* provides a 128K context window and relatively low token cost [6]. *EquivSage* uses default parameters, sets the temperature to 0 to minimize randomness, and applies a three-request mechanism to handle abnormal responses, ensuring stable results.

Dataset: To ensure a comprehensive evaluation, we collected 1,263,683 real-world smart contract source codes from six mainstream blockchains. This dataset extends our previous work [72], which contained 905,948 contracts from six EVM-compatible chains (up to January 2024), by incorporating deployments through July 2025 and adding 357,735 new contracts. Together, these six blockchains represent 83.25% of the total cryptocurrency market capitalization at the time of writing [33], including *Ethereum* (71.43%), *Binance Smart Chain* (BSC) (5.77%), *Arbitrum* (2.63%), *Polygon* (0.92%), *Optimism* (1.26%), and *Avalanche* (1.58%). To our knowledge, this represents the largest publicly available dataset of smart contracts across six EVM-compatible chains, enabling a comprehensive analysis of the real-world distribution of *EVM-Inequivalent Code Smells*.

5.2 RQ1: Performance on large-scale datasets

To evaluate the performance of *EquivSage* on a large dataset and examine the proportion of *EVM-Inequivalent Code Smells* across different blockchains, we analyzed 1,263,683 contracts from six major EVM-compatible blockchains.

Table 7 presents the statistical results of *EquivSage* detection on large-scale datasets on six blockchains. Overall, *EVM-Inequivalent Code Smells* are widely distributed, with an average prevalence of 15.45%. *Avalanche* exhibits the highest proportion (24.95%), followed by *BSC* (18.63%) and *Optimism* (16.61%), while *Polygon* shows the lowest (8.69%). Among the various smell types, *Phishing Contract Address* (PCA), *Time Discrepancy Trap* (TDT), and *Fixed Gas Reentrancy* (FGR) occur most frequently. PCA is the most prevalent, with 79,868 cases in total, concentrated on *BSC* (27,853 cases) and *Ethereum* (24,744 cases). TDT is also widespread, with 40,705 cases, primarily on *BSC* (24,467 cases) and *Ethereum* (12,531 cases). FGR follows with 35,927 cases, again concentrated on *Ethereum* (13,432 cases) and *BSC* (12,788 cases). In contrast, *Inconsistent Precompile Implementation* (IPI) and *Gas Limit Imbalance* (GLI) occur far less frequently, with fewer than 50 cases on most blockchains. These findings indicate that PCA, TDT, and FGR are particularly common across multiple blockchains, reflecting a persistent lack of developer awareness of these issues in reused contracts and underscoring the need for targeted preventive measures.

Table 7. Distribution of EVM-Inequivalent Code Smells across six blockchains

Type	Ethereum	Optimism	BSC	Polygon	Arbitrum	Avalanche	Total / Overall
BHM	9106	49	2893	1091	549	469	14157
BMTA	812	192	167	641	197	250	2259
CCRA	3128	1981	331	2379	517	487	8823
GLI	30	21	43	41	35	11	181
IOI	5667	2755	993	2006	762	1147	13330
IPI	1	3	0	2	1	0	7
PCA	24744	12641	27853	6966	3618	4046	79868
FGR	13432	359	12788	4374	1646	3328	35927
TDT	12531	373	24467	1638	770	926	40705
Prop.(%)	16.03%	16.61%	18.63%	8.69%	9.72%	24.95%	15.45%
Avg. Time (s)	61.08	53.35	56.50	55.85	62.31	66.89	59.33

Table 7 further reports the average per-contract analysis time on each blockchain. The per-chain average runtimes are 61.08 s (Ethereum), 53.35 s (Optimism), 56.50 s (BSC), 55.85 s (Polygon), 62.31 s (Arbitrum), and 66.89 s (Avalanche). The overall runtime is 59.33 s, which denotes the mean per-contract analysis cost over all evaluated contracts rather than an aggregation of chain-level runtimes. The observed cross-chain variation is primarily attributable to differences in contract complexity, the distribution of detected smell categories, and the number of candidate paths that require symbolic-execution-based feasibility checking. Although *Avalanche* and *Arbitrum* incur slightly higher average runtimes than *Optimism* and *Polygon*, the variation remains moderate across all chains. These results complement the prevalence statistics and indicate that *EquivSage* is practical for large-scale, multi-chain deployment.

To characterize distributional changes and assess the methodological improvements over our prior work, we compare the results of *EquivSage* with the corresponding findings reported by *EquivGuard* [72]. The prior study analyzed 905,948 contracts collected up to January 2024, whereas this study analyzes 1,263,683 contracts collected through July 2025. Consequently, the differences between the two sets of results should be interpreted along two dimensions. First, the expanded corpus captures newly deployed contracts and more recent multi-chain reuse practices. Second, the upgraded detector incorporates semantic guidance and path-feasibility validation, enabling it to filter warnings that match syntactic patterns but do not satisfy the semantic definitions of the corresponding smells. Figure 19 summarizes the relative changes between the prior results and those reported in Table 7. Systematic increases, such as the increase of BHM across all six chains, are mainly explained by the longer observation window and the continuing growth of multi-chain reuse. In contrast, systematic decreases in FGR and TDT on several chains, particularly BSC, Arbitrum, Polygon, and Avalanche, indicate improved false-positive filtering: *EquivSage* distinguishes genuine cross-chain semantic risks from syntactic patterns that were previously over-approximated. Mixed trends reflect the joint effect of corpus expansion and detector refinement. PCA provides a representative example. Its occurrence increases on Ethereum, Optimism, and Avalanche, suggesting that the expanded corpus introduces additional PCA instances on these chains; however, it decreases on BSC from 35,018 to 27,853 and on Polygon from 7,102 to 6,966. These decreases are attributable to detector refinement, because *EquivSage* further checks whether suspicious address uses can induce cross-chain phishing semantics under the surrounding control dependencies, data dependencies, and feasible execution paths.

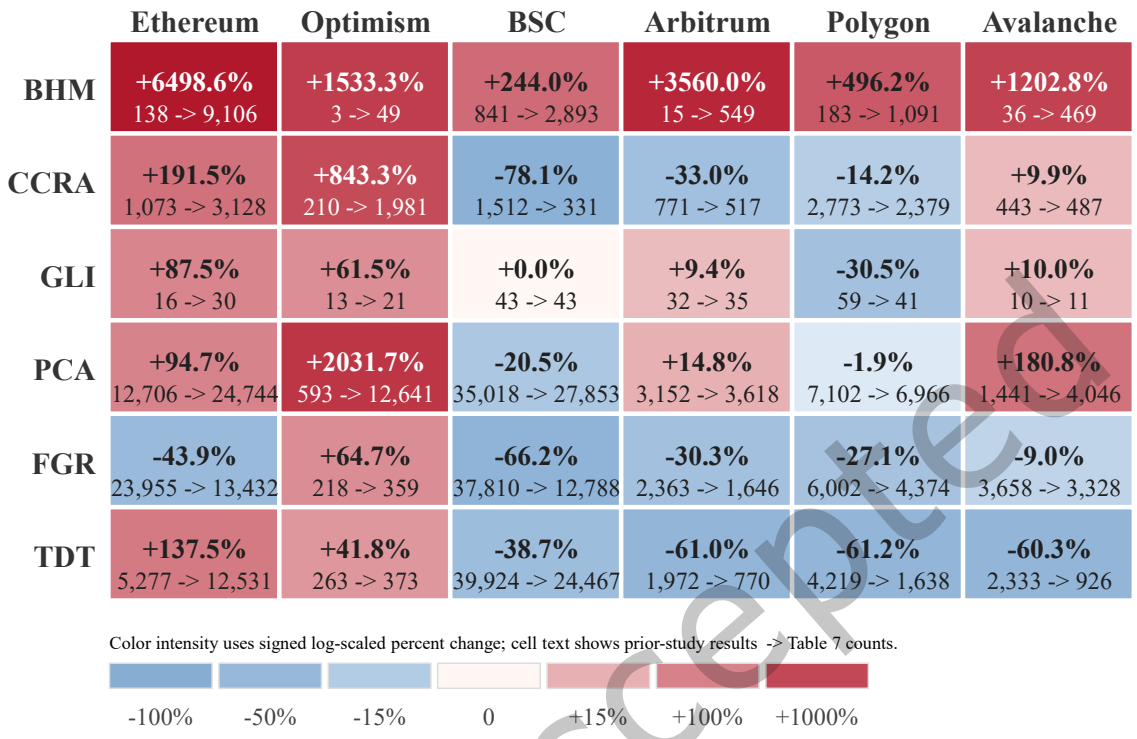


Fig. 19. Relative changes between the prior-study results and the results in Table 7. Each cell reports the percentage change and the corresponding count shift from the prior study to this study. Red cells indicate increases, blue cells indicate decreases, and color intensity is computed using a signed log-scaled percentage change to reduce the visual dominance of extremely large ratios caused by small baselines.

Answer to RQ1: The analysis reveals that *EVM-Inequivalent Code Smells* are widespread, with an average occurrence rate of 15.45%. *Phishing Contract Address* (PCA), *Time Discrepancy Trap* (TDT), and *Fixed Gas Reentrancy* (FGR) are prevalent across multiple blockchains, underscoring the necessity of raising developer awareness of these risks.

5.3 RQ2: Detection Effectiveness

Ground-truth Construction. To evaluate the detection effectiveness of *EquivSage*, we constructed the ground truth through a three-step procedure: statistically guided sampling, independent manual annotation, and consensus-based reconciliation. Following prior work [72], we sampled 738 tool-reported positive cases and 97 tool-unreported negative cases from the detection results using a 95% confidence level and a confidence interval width of 10 percentage points.

For each sampled case, the same two authors served as annotators and independently inspected the reported smell type, location, root cause, and surrounding contract context. Both annotators are Ph.D. students specializing in smart contract security; each has published at least three papers on smart contract security in top-tier software engineering or security venues, and both have practical experience in smart contract vulnerability analysis

and security auditing. A tool-reported positive case was labeled as TP only when the reported location and cause satisfied the corresponding *EVM-Inequivalent Code Smell* definition; otherwise, it was labeled as FP. For a tool-unreported negative case, the annotators inspected the same code context to determine whether a smell instance had been missed. If a missed instance was found, the case was labeled as FN; otherwise, it was labeled as TN. The annotation followed the type-specific decision criteria summarized in Section 4.6. During independent annotation, the two annotators disagreed on 27 sampled cases, yielding Cohen's $\kappa = 0.853$, which indicates substantial agreement. All disagreements were resolved through consensus discussion by re-examining the contract code, detection report, smell definition, and surrounding execution context. After reconciliation, the final manually verified dataset contained 730 ground-truth positive cases and 105 ground-truth negative cases. Compared with *EquivSage*'s outputs, these cases correspond to 721 true positives, 17 false positives, 9 false negatives, and 88 true negatives. We release the sample-level annotation records in the artifact.

Table 8. Precision and recall evaluation of *EquivSage* across different code smell types

Type	BHM	BMTA	CCRA	GLI	IOI	IPI	PCA	FGR	TDT
#Sampled	96	92	95	64	96	7	96	96	96
#TP	91	92	94	64	96	7	96	90	91
#FP	5	0	1	0	0	0	0	6	5
#FN	0	0	7	0	0	2	0	0	0
Precision(%)	94.79%	100.00%	98.95%	100.00%	100.00%	100.00%	100.00%	93.75%	94.79%
Recall(%)	100.00%	100.00%	93.07%	100.00%	100.00%	77.78%	100.00%	100.00%	100.00%

False Positive Analysis. To quantify precision, we analyzed the tool-reported positive samples by smell category. For each category, precision was computed as $Precision = \frac{TP}{TP+FP}$. The overall precision was then computed as a category-size-weighted average, i.e., $Precision(Overall) = \frac{\sum_{i=1}^{n} p_{c_i} \times |c_i|}{\sum_{i=1}^{n} |c_i|}$, where p_{c_i} denotes the precision for code smell i and $|c_i|$ denotes the number of sampled cases in that category [79]. As shown in Table 8, *EquivSage* achieves an overall precision of 97.70%, demonstrating that most reported *EVM-Inequivalent Code Smells* correspond to manually validated smell instances.

Our analysis shows that false positives mainly stem from the limited ability of LLMs to reason about intermediate program states without execution feedback. In these cases, the LLM does not simply fail to identify suspicious code patterns. Rather, it often detects a risk-related syntactic pattern correctly, but fails to preserve the semantics of intermediate transformations that normalize or sanitize the initial risk before the final path constraint is evaluated [21]. Figure 20 illustrates this pattern. The `withdraw()` function contains `block.number`, which is a relevant syntactic signal for a potential TDT risk. However, the function does not directly interpret block height as elapsed time. It first computes `blocksPassed`, converts this value into chain-specific elapsed time through `chainBlockTime[block.chainid]`, and then incorporates `GRACE_PERIOD` into the final `require` condition to tolerate minor fluctuations in block production time. This sequence mitigates the initial risk by normalizing block height into chain-aware elapsed time and adding an explicit tolerance margin. The false positive occurs because the LLM fails to retain and reason over this intermediate state transformation, instead relying primarily on the presence of `block.number` in a temporal check.

False Negative Analysis. To assess completeness, we extend Table 8 with recall-related statistics and compute recall as $Recall = \frac{TP}{TP+FN}$. The table reports the number of sampled positive reports, TP, FP, FN, precision, and

```
uint256 public constant LOCK_PERIOD = 7 days;
uint256 public constant GRACE_PERIOD = 1 hours; // Tolerance window
mapping(address => uint256) public depositBlock;
mapping(uint256 => uint256) public chainBlockTime; // chainId -> avg block time
constructor() {
    chainBlockTime[1] = 12;   // Ethereum average block time ~12s
    chainBlockTime[56] = 3;   // BSC average block time ~3s
    chainBlockTime[137] = 2;  // Polygon average block time ~2s
}
...
function withdraw() external {
    uint256 blocksPassed = block.number - depositBlock[msg.sender];
    uint256 secondsPassed = blocksPassed * chainBlockTime[block.chainid];
    // Apply tolerance window to absorb boundary fluctuations
    require(secondsPassed + GRACE_PERIOD >= LOCK_PERIOD, "Funds are still locked!");
    ...
}
```

Fig. 20. Example of a false positive

recall for each of the nine smell types. Overall, *EquivSage* achieves a recall of 98.77%. The false negatives are concentrated in two smell types rather than distributed broadly across categories: CCRA has 7 false negatives with a recall of 93.07%, and IPI has 2 false negatives with a recall of 77.78%. The remaining seven smell types have no false negatives and achieve 100.00% recall. This distribution suggests that the recall limitation of *EquivSage* is not a general weakness across smell categories, but mainly arises in scenarios that require recognizing low-level invocation patterns. Manual inspection shows that the missed CCRA cases involve non-standard signature recovery logic, whereas the missed IPI cases involve precompiled-contract invocations implemented through inline assembly or heavily customized low-level code. In such cases, the relevant operations may not appear as standard AST-level call patterns. Consequently, Phase 1 slicing may fail to identify the corresponding *Key Code Blocks*, preventing the subsequent LLM-assisted analysis from being triggered. Prior work [18] has also observed that manual inline assembly lacks generality and that its complex logic increases the difficulty of understanding and maintaining the code, which in turn reduces the feasibility of direct reuse.

Answer to RQ2: *EquivSage* achieves a precision of 97.70% and a recall of 98.77% in detecting *EVM-Inequivalent Code Smells*, demonstrating high detection reliability.

5.4 RQ3: Ablation Experiment

We conducted ablation experiments to quantify the individual contributions of two core mechanisms in *EquivSage*: *LLM Assistance* (LLMA) and *Symbolic Execution Verification* (SEV). The ablation boundary is defined at the mechanism level rather than the phase level, because the two analysis phases are not independently replaceable. In particular, *Phase 1* uses LLMA to identify task-relevant variables and derive sliced code from *Key Code Blocks* and the I-PDG, whereas *Phase 2* takes the resulting slices as input to recognize sanitization evidence and perform taint analysis. Consequently, removing LLMA from only one phase would simultaneously alter the input distribution, intermediate representations, or semantic assumptions of the other phase, thereby confounding the interpretation of the ablation result. To avoid this ambiguity, we evaluate LLMA as a unified semantic-guidance mechanism spanning both phases, and evaluate SEV independently as a path-feasibility validation mechanism.

Following this mechanism-level design, we evaluate four configurations. *Full Setup* denotes the complete *EquivSage* pipeline with both LLMA and SEV enabled. *No LLM Assistance* removes LLMA while maintaining an

executable analysis workflow by replacing it with the fixed *Domain-Specific Patterns* inherited from our prior work, *EquivGuard* [72]. *No Symbolic Execution* retains LLMA but disables SEV, thereby omitting path-feasibility validation. *Neither* disables both mechanisms, providing a lower-bound configuration without semantic guidance or path validation. These configurations are designed to disentangle the effect of semantic guidance from that of path verification while preserving the comparability of the overall workflow. For a controlled evaluation, all configurations are applied to the same labeled RQ2 dataset, which consists of 730 positive samples and 105 negative samples that were manually inspected and verified, with a fixed timeout of 150 seconds for each run.

Table 9. Detection performance under ablation settings and standalone LLM baselines

	Full Setup	No LLM Assis-tance	No Symbolic Execution	Neither	DeepSeek V3.1	Gemini 3.1 Pro	GPT-5.4	Claude Opus 4.6
#TP	721	406	234	62	54	107	194	123
#FP	17	51	77	105	692	548	470	617
#FN	9	324	496	668	13	83	87	7
#TN	88	54	28	0	76	97	84	88
Precision (%)	97.70%	88.84%	75.24%	37.13%	7.24%	16.34%	29.22%	16.62%
Recall (%)	98.77%	55.62%	32.05%	8.49%	80.60%	56.32%	69.04%	94.62%
F1-score (%)	98.23%	68.41%	44.96%	13.82%	13.28%	25.33%	41.06%	28.28%
Avg. Time (s)	62.07	135.02	47.16	19.1	24.662	40.482	6.057	125.088
Total Cost ($)	2.82	0	2.82	0	7.18	49.93	59.53	88.76
# UNKNOWN	6	571	–	–	–	–	–	–

Effectiveness Analysis. Table 9 reports the detection performance of the four *EquivSage* configurations and the standalone LLM baselines. The *Full Setup* achieves the strongest overall performance, with 97.70% precision, 98.77% recall, and a 98.23% F1-score (#TP = 721, #FP = 17, #FN = 9). Removing either LLMA or *Symbolic Execution Verification* (SEV) substantially degrades performance, but the resulting error patterns differ. When LLMA is disabled, the F1-score decreases to 68.41%, recall drops to 55.62%, and false negatives increase to 324. This result indicates that fixed domain rules retain partial detection capability, but do not provide sufficient semantic generalization for non-standard implementations or implicit contextual constraints. When SEV is disabled, the F1-score further decreases to 44.96%, recall drops to 32.05%, and false negatives increase to 496, suggesting that LLM-guided candidates cannot be reliably promoted to confirmed detections without path-feasibility validation. When both mechanisms are removed, the *Neither* configuration performs worst, with only 37.13% precision, 8.49% recall, and a 13.82% F1-score. Overall, the ablation results indicate that LLMA and SEV contribute complementary capabilities: LLMA improves semantic coverage during slicing and taint analysis, whereas SEV improves detection reliability by validating feasible execution paths.

To evaluate the role of program-analysis constraints in LLM-assisted detection, we conduct a complementary LLM-only experiment using standalone LLM baselines without explicit program-analysis support. Specifically, we instantiate pure LLM-based analyzers with *DeepSeek-V3.1*, *Gemini 3.1 Pro*, *GPT-5.4*, and *Claude Opus 4.6*. Their F1-scores are 13.28%, 25.33%, 41.06%, and 28.28%, respectively, which are all substantially below the 98.23% F1-score achieved by the *Full Setup*. Although some standalone LLMs achieve high recall, e.g., 80.60% for *DeepSeek-V3.1* and 94.62% for *Claude Opus 4.6*, their recall is obtained at the cost of many false positives, including 692 FP cases for *DeepSeek-V3.1* and 617 FP cases for *Claude Opus 4.6*. These results indicate that unconstrained LLM reasoning can cover many suspicious cases, but it tends to over-approximate program semantics in the absence of explicit dependency tracking, taint propagation, and path-level feasibility validation. Therefore, the empirical advantage of *EquivSage* stems from the integration of LLM-based semantic guidance with program-analysis mechanisms, rather than from fixed-pattern analysis or standalone LLM reasoning alone. The performance variation across

standalone models (F1-scores ranging from 13.28% to 41.06%) confirms that model choice substantially affects standalone detection performance. However, this variation does not proportionally transfer to the integrated pipeline: the I-PDG-based taint analysis and symbolic execution impose structural constraints that bound the influence of intermediate LLM outputs on the final detection decision, reducing the pipeline's sensitivity to the specific model backend. In addition, *DeepSeek-V3.1* incurs substantially lower per-token cost than the other evaluated models (total cost of $7.18 for 835 samples vs. $49.93–$88.76 for pure-LLM analysis with other backends), making it better suited for large-scale experiments that require repeated LLM invocations across over million contracts.

To further quantify the contribution of LLMA, we conduct a no-LLM controlled experiment by comparing *EquivSage* with the fixed-pattern analysis implemented in *EquivGuard*. As shown in Figure 21, the 406 true positives reported by *EquivGuard* are fully subsumed by *EquivSage*; in addition, *EquivSage* detects 315 true positives that are missed by *EquivGuard*. This result suggests that LLMA broadens the effective coverage of fixed-pattern analysis, especially when the evidence for an *EVM-Inequivalent Code Smell* is not encoded in canonical syntactic forms, such as standardized variable names, local intra-contract data flows, or explicit sanitization routines. The overlap of false negatives is consistent with this interpretation: the 9 false negatives of *EquivSage* are all contained in the 324 false negatives of *EquivGuard*, indicating that the remaining missed cases are also challenging for fixed-pattern localization and are not newly introduced by LLMA. For false positives, *EquivSage* removes 41 false alarms produced by *EquivGuard* by considering contextual constraints that are not captured by rigid syntactic patterns, while introducing 7 new false positives in cases involving multi-step state transformations, implicit constraints, or delayed sanitization logic. Overall, this comparison indicates that LLMA mainly improves semantic coverage and substantially reduces false negatives caused by rigid pattern matching, with an additional effect of mitigating some pattern-induced false alarms; residual errors remain when the relevant equivalence condition depends on subtle state changes or late sanitization effects.

The solver outcomes provide a complementary, mechanism-level explanation for the contribution of LLMA. In Table 9, the *# UNKNOWN* row reports the number of path-constraint queries for which Z3 returns UNKNOWN, whereas TP, FP, FN, and TN are reported as sample-level detection outcomes. Under the *Full Setup*, only 6 solver queries are classified as UNKNOWN; under *No LLM Assistance*, this number increases to 571. The *No Symbolic Execution* and *Neither* settings do not invoke Z3, and therefore solver-undecided outcomes are not applicable to them. We conservatively treat UNKNOWN queries as inconclusive: they are recorded separately, excluded from confirmed detection results, and never used as evidence for a confirmed detection unless a solver-confirmed SAT path is obtained. The substantially smaller number of UNKNOWN queries in the *Full Setup* suggests that LLMA

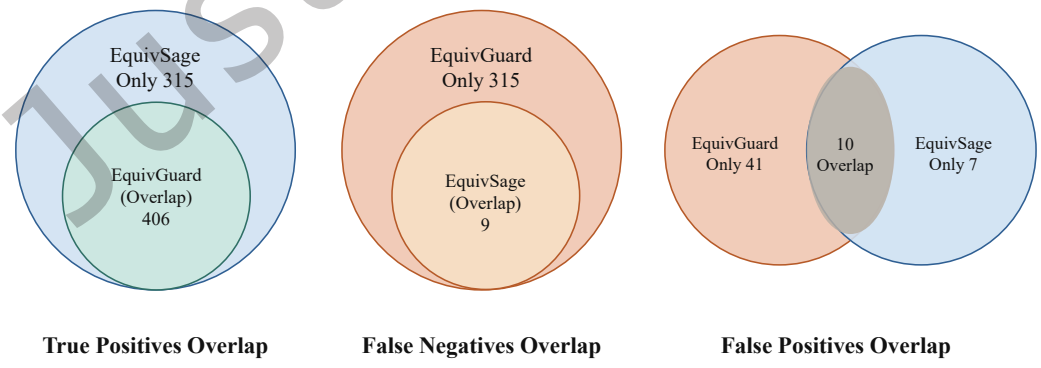


Fig. 21. Venn diagram of true positives, false negatives, and false positives detected by *EquivSage* and *EquivGuard*.

reduces solver burden by pruning irrelevant dependencies and low-relevance candidate paths before symbolic execution.

Efficiency and Cost Analysis. Table 9 further reports the average detection time and LLM usage cost, thereby characterizing the practical efficiency of each configuration. The *Full Setup* completes detection in 62.07 s on average while achieving the best accuracy. Disabling SEV reduces the average time to 47.16 s by eliminating path traversal and constraint solving; however, this reduction is accompanied by substantial degradation in both precision and recall. In contrast, disabling LLMA leads to the longest average time, 135.02 s, despite its lower accuracy. This result can be explained by the fact that *No LLM Assistance* does not eliminate domain guidance entirely; instead, it retains the executable workflow based on the domain-specific patterns inherited from *EquivGuard*, while removing the contextual semantic pruning introduced by LLMA. Consequently, more irrelevant dependencies and candidate paths are propagated to symbolic execution, increasing both path-exploration overhead and the number of solver-undecided queries. The *Neither* configuration achieves the shortest average time, 19.1 s, but also yields the weakest detection performance, indicating that its apparent efficiency gain is obtained at the expense of accuracy and coverage. Regarding monetary cost, the *Full Setup* processes 835 manually verified samples with a total LLM cost of $2.82, corresponding to approximately $0.0034 per sample. *No LLM Assistance* and *Neither* incur no LLM cost, whereas *No Symbolic Execution* incurs the same LLM cost as the *Full Setup* because it preserves the same LLM-assisted analysis steps. For pure-LLM analysis, the total costs of *DeepSeek-V3.1*, *Gemini 3.1 Pro*, *GPT-5.4*, and *Claude Opus 4.6* are $7.18, $49.93, $59.53, and $88.76, respectively. These results suggest that the targeted use of LLMs in *EquivSage* incurs modest monetary overhead, whereas pure-LLM analysis can be both less accurate and more costly depending on the selected backend.

Answer to RQ3: *LLM Assistance* improves semantic coverage by guiding slicing and taint analysis toward contextually relevant code, while *Symbolic Execution Verification* improves reliability by confirming path feasibility. Their integration enables *EquivSage* to achieve accurate, reliable, and cost-effective detection of *EVM-Inequivalent Code Smells*.

5.5 RQ4: Prevalence and Distribution of *EVM-Inequivalent Code Smells* in Multi-Chain Reused Contracts

To investigate the prevalence and distribution of *EVM-Inequivalent Code Smells* in multi-chain reused contracts, we analyzed 69,451 *Ethereum* contracts identified in RQ1 and traced their reuse across five EVM-compatible blockchains: *Binance Smart Chain*, *Arbitrum*, *Polygon*, *Optimism*, and *Avalanche*. Specifically, following the *Clone Type I* definition proposed by Roy et al. [57]. in software clone detection, reuse is defined as identical source code produced by direct *"copy-and-paste"* migration. Accordingly, we verified multi-chain reused contracts by comparing whether contracts deployed at the same address across different chains shared identical bytecode hashes. As shown in Figure 22, the analysis proceeded in three steps: (1) we applied the *EquivSage* tool to scan 433,375 *Ethereum* contracts and identified 69,451 containing *EVM-Inequivalent Code Smells*; (2) for each address, we queried the five chains via RPC interfaces to retrieve deployed bytecode, recording the hash of bytecode if available or assigning a null value otherwise; (3) we compared these hashes with the corresponding *Ethereum* contract's hash and classified identical matches as multi-chain reuse. Using this method, we identified 2,259 multi-chain reused contracts and subsequently examined the distribution.

Table 10 shows that the distribution of *EVM-Inequivalent Code Smells* in multi-chain reused contracts is highly uneven. *Phishing Contract Address* (PCA) is the most common type, accounting for 45.22% of all cases, nearly half of the dataset, and indicating strong prevalence and reuse in practice. The second most frequent type is *Cross-Chain Replay Attack* (CCRA) at 26.81%; together, PCA and CCRA dominate the distribution. *Fixed Gas Reentrancy* (FGR) and *Inconsistent Opcodes Implementation* (IOI) represent 11.90% and 10.35%, respectively, and are moderately frequent. Other types, including *Time Discrepancy Trap* (TDT), *Broken msg.sender == tx.origin*

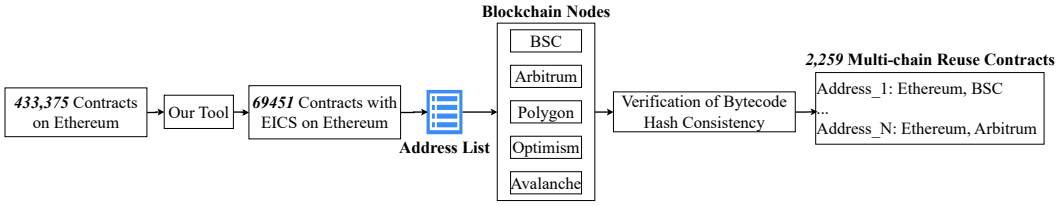


Fig. 22. Analysis Process of Multi-Chain Contract Reuse

Assumption (BMTA), *Block Height Misalignment* (BHM), *Inconsistent Precompile Implementation* (IPI), and *Gas Limit Imbalance* (GLI), each account for less than 3%, making them rare. In terms of code size and complexity, most types have an average number of *lines of code* (LOC) between 1700–1900, with the number of functions (#Func) typically ranging from 170–215. This shows that such smells are not confined to small or simple contracts but also appear in medium and large scale. The complexity metric (MaxAvgCC) generally ranges from 6.2–7.7, with BMTA and GLI showing the highest values (7.68 and 7.00), likely due to their specific logic. From an economic perspective, PCA contracts stand out, holding the highest value of chain assets (average 44,998,906 USD) and the highest transaction volume (#Txn = 637,470). In summary, *EVM-Inequivalent Code Smells* in multi-chain reused contracts are highly concentrated in a few types (notably PCA and CCRA) with significant economic impact, and should still be addressed by detection and defense mechanisms.

Table 10. Statistics of different types of EVM-Inequivalent Code Smells in multi-chain reused contracts

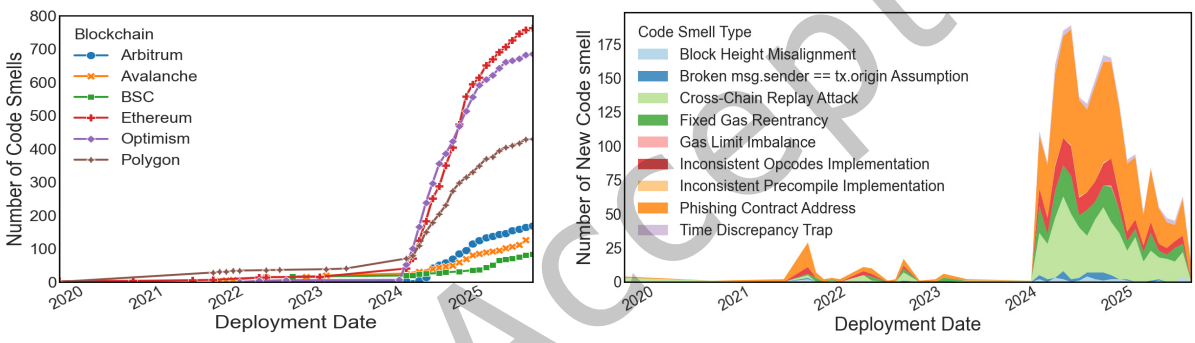
Type	Prop.(%)	LOC (avg)	#Func (avg)	MaxAvgCC	Asset ($)	#Txn
CCRA	26.81%	1772.85	189.01	6.66	771,972.83	342,003
TDT	2.52%	1668.88	167.96	6.61	1,632.28	32,244
FGR	11.90%	1923.09	215.09	6.97	1,814,059.29	219,059
BHM	1.28%	1741.46	173.54	6.46	185.65	63,033
PCA	45.22%	1707.15	183.82	6.45	44,998,906.02	637,470
GLI	0.04%	1897.00	97.00	7.00	90.30	0
BMTA	1.64%	2035.08	216.78	7.68	16,090.25	15,851
IOI	10.35%	1854.14	202.12	6.73	1,476,851.65	195,445
IPI	0.22%	1396.80	142.80	6.20	0.00	119

To further investigate the prevalence of different types of *EVM-Inequivalent Code Smells* across blockchains, we examined the correlation between the deployment dates, code smell counts, and types of reused contracts across blockchains, as summarized in Figure 23. This analysis also provides insight into how developers' awareness of these issues has evolved over time.

As shown in Figure 23a, the time-series distribution of reused contracts with *EVM-Inequivalent Code Smells* varies significantly across blockchains. *Ethereum* has experienced a sharp increase since 2024, peaking at nearly 800 cases in 2025, making it the platform with the highest number of code smells. *Polygon* shows a similar growth trend but with a slightly lower peak of about 600 cases, indicating that it is also a major concentration of multi-chain reuse risks. In contrast, *BSC*, *Arbitrum*, *Optimism*, and *Avalanche* exhibit steady growth year-on-year.

Although their totals remain below those of *Ethereum* and *Polygon*, the consistent increase suggests that code smells on these platforms are becoming an increasingly important concern for multi-chain security.

As shown in Figure 23b, the distribution by type shows clear differences in both frequency and timing. From 2020 to 2023, new cases for each code smell type remained low and stable, suggesting that early multi-chain reused contracts had a consistent pattern of security defects. From 2024 onward, several types, including *Phishing Contract Address* (PCA), *Cross-Chain Replay Attack* (CCRA), and *Inconsistent Opcodes Implementation* (IOI), increased sharply and accounted for most of the overall growth. This rise is likely related to the expansion of multi-chain deployments, the growing complexity of cross-chain interactions, and the gaps in developer security awareness. It is noteworthy that some low-frequency types (such as *Gas Limit Imbalance* (GLI), *Block Height Misalignment* (BHM), and *Inconsistent Precompile Implementation* (IPI)), although accounting for only a small proportion overall, still exhibit periods of concentrated occurrence within specific time intervals. This suggests that their appearance may be related to particular business logic or on-chain events. In addition, the temporal distribution shows that the growth trends of the high-frequency types are synchronized across multiple blockchains, the effect being most pronounced on *Ethereum*, *Polygon* and *Optimism*. This may reflect the cross-platform propagation of *EVM-Inequivalent Code Smells* under multi-chain reuse patterns.



(a) Deployment Dates and Code Smell Counts of Reused Contracts Across Blockchains

(b) Deployment Dates and Code Smell Counts of Reused Contracts by Type

Fig. 23. Comparison of deployment dates and code smell counts of reused contracts across blockchains and by type

Answer to RQ4: *EVM-Inequivalent Code Smells* in multi-chain reused contracts are highly concentrated. Specific types such as PCA, CCRA and IOI, dominate on *Polygon* and *Optimism*. Their numbers have grown rapidly since 2024, making them an important security risk that requires urgent action from multi-chain developers.

6 Discussion

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

Through large-scale detection, *EquivSage* 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 122 [3], while the actual chainId of the blockchain deployed was 1284. This caused the failure of the verification

[3]https://moonscan.io/address/0x818ec0a7fe18ff94269904fced6ae3dae6d6dc0b#code#L306

mechanism intended to prevent replay attacks. This threat continuously affects assets on other chains where the same contracts are reused.

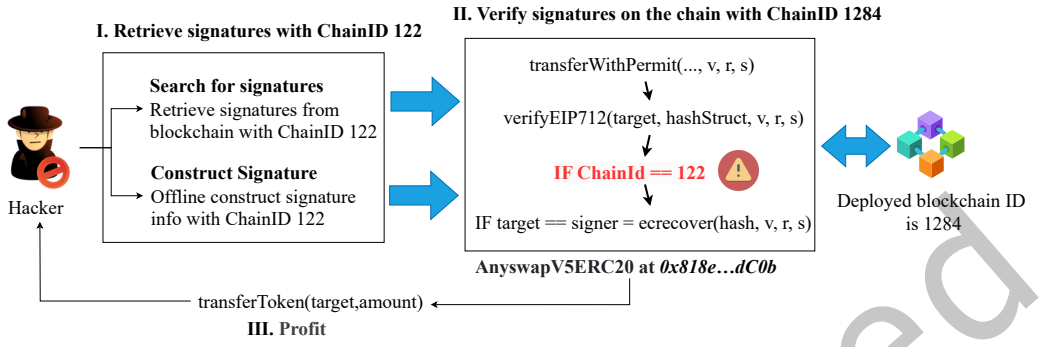


Fig. 24. CCRA Exploitation in the *Multichain*'s contract

As shown in Figure 24, 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 is successful is due to a check using the incorrect `ChainID` 122 in the signature verification logic. (III) Profit. Attackers transfer tokens to gain profit.

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. *EquivSage* promptly identified and reported this code smell.

Beyond aggregate detection metrics, this case study demonstrates the practical relevance of *EquivSage* in exposing *EVM-Inequivalent Code Smells* that may induce security-relevant behavioral divergence in deployed multi-chain systems. In the CCRA case from the *Multichain* project, the hard-coded `chainId` embedded in reused contract logic is inconsistent with the actual deployment chain, thereby invalidating the intended signature verification condition and placing approximately $28.52K in assets at risk. The large-scale results further suggest that such risks are not anecdotal: *EVM-Inequivalent Code Smells* are observed across reused contracts on six major EVM-compatible blockchains, with non-trivial type distributions and temporal trends. We emphasize that a detected code smell should not be directly interpreted as a confirmed exploit or realized financial loss; rather, it serves as an early-warning signal for reused contracts that may rely on chain-specific assumptions and therefore warrant further security inspection.

6.2 Implications

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

For Auditors. Auditors can use *EquivSage* 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 [43] resulted in a $20 million loss when Ethereum-safe code was reused on *Optimism*. *EquivSage* helps auditors analyze multi-chain execution inconsistencies in reused contracts and provides comprehensive security guidance based on different blockchain designs (e.g., *Gas Mechanisms* and *Consensus Protocols*).

For Developers. *EquivSage* helps developers avoid negative impacts due to misunderstandings of *direct copying of smart contracts for multi-chain deployments*. While multi-chain reuse saves costs, it can introduce *EVM-inequivalent code smells*, creating security threats. *EquivSage* 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 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 originating 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 developers.

6.3 Threats to Validity

Internal Validity. The first internal threat concerns the use of LLMs in program analysis. LLMs may hallucinate, exhibit model-dependent variability, and generate inconsistent intermediate semantic guidance under different model configurations. They may also fail to capture subtle runtime state changes, potentially leading to false positives or false negatives. In addition, potential data leakage cannot be completely ruled out because LLM pretraining corpora are not fully observable. To mitigate these risks, *EquivSage* uses the LLM only to provide intermediate semantic guidance for slicing and sanitized-variable identification, rather than as the final detector. Candidate reports are further validated through I-PDG-based taint analysis and symbolic execution, thereby reducing the influence of unreliable LLM outputs on confirmed detection results.

The second internal threat stems from manual screening and annotation. The identification of reports, classification of code smells, and assignment of TP/FP/FN/TN labels all involve human judgment, which may introduce subjectivity or confirmation bias. To mitigate this threat, two annotators with expertise in smart contract security independently reviewed the empirical records and sampled cases according to unified screening, classification, and annotation criteria. For each case, they examined the reported code smell type, location, root cause, and surrounding contract context before assigning labels. Disagreements were measured using inter-annotator agreement and resolved through consensus discussion. We also release the corresponding decision logs and annotation records to support auditability.

The third internal threat concerns the fixed-pattern localization of *Key Code Blocks* in Phase 1. Non-standard low-level implementations may encapsulate critical operations, such as precompiled contract calls or `ecrecover()`-related logic, within highly customized inline assembly code. In such cases, Phase 1 may fail to trigger the subsequent LLM-assisted analysis. Our false-negative analysis shows that this issue mainly affects CCRA and IPI, accounting for 9 missed cases among 730 ground-truth positive samples. Therefore, it may reduce recall for highly customized low-level code, although its observed impact on the reported results remains limited.

External Validity. External validity threats primarily concern the generalizability of the code smell definitions and the completeness of the collected evidence. To ensure that the definitions reflect practical concerns of developers and auditors, we derived them from multiple sources, including 1,379 audit reports, 823 bug bounty reports, and 326 *Stack Overflow* posts related to contract reuse. Nevertheless, relevant cases may still be underrepresented if they use different terminology or appear outside the selected sources. To reduce this risk, we used an extended set of related search terms, including "*EVM*", "*EVM Equivalent*", and "*EVM Compatible*", and conducted two rounds

of manual review to remove irrelevant records and improve coverage. In addition, our large-scale evaluation covers real deployed contracts from six major EVM-compatible blockchains, which improves ecological validity.

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. [46] 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* [38] 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 [22]. Zhang et al. [81] through a large-scale analysis of *Ethereum* transactions, smart contracts, and *StackExchange* posts, found that developers face five 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. [79] 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. [23] analyzed 146,452 open-source *Ethereum* contracts, finding widespread code reuse with ERC20 tokens the most frequently reused. Sun et al. [61] 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 patterns of frequent reuse to guide the development of secure contracts. Pierro et al. [53] compared the major smart contract corpora, noting that only a small portion reused code from the secure *OpenZeppelin* repository [51]. They recommended leveraging the *OpenZeppelin* Solidity Library to improve contract security. Huang et al. [42] constructed a semantic *Code Knowledge Graph* to uncover unknown factors in smart contract reuse, effectively improving code recommendation accuracy and developer efficiency.

Differences. First, to the best of our knowledge, this is the first study to systematically investigate inconsistent execution of reused contracts across EVM-compatible blockchains. Although reusing *Solidity* contracts across platforms may introduce new attack surfaces, *EVM-Inequivalent Code Smells* remain insufficiently studied. Second, our analysis triangulates evidence from *Stack Overflow* posts, bug bounty reports, and security audit reports, enabling us to capture developers' practical concerns and characterize real-world manifestations of these issues. Third, we propose a detection method that integrates LLM-based contextual semantic analysis with program analysis. Existing static-analysis-based approaches typically rely on fixed patterns or manually specified rules. *EquivSage* also incorporates expert-defined domain patterns, but differs in how these patterns are operationalized. Rather than depending solely on rigid syntactic matching, *EquivSage* uses the LLM to interpret sliced code contexts and identify relevant variables, dependencies, and sanitization conditions under the guidance of these patterns. Thus, the contribution of *EquivSage* does not lie in eliminating expert knowledge, but in combining explicit domain knowledge, LLM-based contextual analysis, and subsequent program-analysis-based verification.

7.3 Compared with Our Previous Work

This study extends our prior research on *EVM-Inequivalent Code Smells*, providing a more comprehensive treatment of their definition (Section 3), methodological design (Section 4), and experimental evaluation (Section 5).

To better capture the *EVM-Inequivalent* issues that developers encounter when reusing contract code, we broadened the scope of our empirical study. Bug bounty reports, produced by independent security researchers,

typically include reproducible scripts or detailed analyses that undergo peer review, thereby serving as a valuable complement for uncovering emerging vulnerabilities. In addition to 1,379 public audit reports from 30 security teams and 326 related *Stack Overflow* posts, we further collected 823 reports from leading smart contract security bounty platforms, including *Code4Rena* [1], *Immunefi* [3], *Sherlock* [4], and *Codehawks* [2] (Section 3.1). After filtering and relevance analysis, 336 source records related to *EVM-Inequivalent* issues in contract reuse were retained, from which 340 code-smell instances were extracted (Section 3.2). Using the *Open Card Sorting* method, we classified these into nine types of *EVM-Inequivalent Code Smells*. Compared with our earlier work, we introduced three new types, namely BMTA, IOI, and IPI, which arise from inconsistencies in EIP upgrade support across different blockchains (Section 3.3).

To improve detection effectiveness, we developed the *EquivSage* tool, which integrates the contextual semantic understanding of LLMs with program analysis. Different from our previous tool *EquivGuard*, which relied on fixed patterns, *EquivSage* leverages LLMs to guide code slicing and static taint analysis. Using *Tier of Thought* (ToT) prompts, the LLM incrementally interprets context, extracts key functions and variables, and preserves semantic coherence (Sections 4.3 and 4.4). This reduces false positives and false negatives caused by context loss in traditional program analysis. In addition, *EquivSage* incorporates symbolic execution to explore suspicious paths and verify their feasibility (Section 4.5), further enhancing detection accuracy.

For large-scale evaluation, we collected 1,263,683 smart contract source codes from six mainstream blockchains. Compared with our previous dataset (up to January 2024), this study extends the deployment period to July 2025 and adds 357,735 new contracts. Using this dataset, we evaluated *EquivSage* at scale (Section 5.2), assessed its detection effectiveness (Sections 5.3 and 5.4), and analyzed the distribution of multi-chain reused contracts as well as the prevalence of *EVM-Inequivalent Code Smells* (Section 5.5). The results provide practical insights for strengthening security practices in multi-chain development.

8 Conclusion

In this paper, we conducted an empirical study on *EVM-Inequivalent Code Smells*, focusing on inconsistent execution of *Solidity* contracts when reused on EVM-compatible blockchains. By examining 1,379 security audit reports, 326 *Stack Overflow* posts, and 823 bug bounty reports, we systematically identify and define nine types of *EVM-Inequivalent Code Smells*. These smells, which cause inconsistent execution in multi-chain reuse, pose significant risks to secure and reliable contract deployment. To automate the detection of *EVM-Inequivalent Code Smells*, we then designed *EquivSage*, a tool that integrates LLMs with program analysis. *EquivSage* leverages LLMs to guide code slicing and taint analysis, using *Tier of Thought* (ToT) prompts to incrementally interpret context, extract key functions and variables, and preserve semantic coherence. In addition, it incorporates symbolic execution to verify path feasibility. We evaluated *EquivSage* on 1,263,683 contracts from six major blockchains, achieving a detection precision of 97.70%. Our experiments further revealed that, on average, 15.45% of contracts contained at least one such code smell, underscoring their prevalence and highlighting the need for developers to address inconsistencies in multi-chain contract reuse.

9 Data Availability

We have made *EquivSage*'s source code, empirical study materials, experimental datasets, results, and reproduction instructions publicly available at https://zenodo.org/records/20512279. The artifact also includes key metadata, manual screening and annotation records, and runtime logs to support the traceability and auditability of our empirical results and evaluation metrics.

References

[1] 2026. Code4rena: Keeping high severity bugs out of production. https://code4rena.com/. Accessed: 2026-05-28.
[2] 2026. Cyfrin CodeHawks: Competitive Smart Contract Audits. https://codehawks.cyfrin.io/. Accessed: 2026-05-28.

[3] 2026. Immunefi: Unified Web3 Security Platform. https://immunefi.com/. Accessed: 2026-05-28.

[4] 2026. Sherlock: Bug Bounty Platform for Web3 Security & Rewards. https://sherlock.xyz/bug-bounties. Accessed: 2026-05-28.

[5] 0x52. 2026. 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.

[6] DeepSeek AI. 2026. DeepSeek API Documentation: Your First API Call. https://api-docs.deepseek.com/. Accessed: 2026-05-28.

[7] DeepSeek AI. 2026. DeepSeek: General AI Foundation Models and Technologies. https://www.deepseek.com/. Accessed: 2026-05-28.

[8] Alex Beregszaszi (@axic), Jacques Wagener (@jacqueswww). 2018. EIP-1380: Reduced gas cost for call to self. https://eips.ethereum.org/EIPS/eip-1380.

[9] Arbitrum. 2026. Arbitrum — The Future of Ethereum. https://arbitrum.io/.

[10] AuditBase. 2026. Consider using block.number instead of block.timestamp. https://detectors.auditbase.com/blocknumber-vs-timestamp-solidity.

[11] AuditOne. 2026. 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.

[12] David Binkley, Nicolas Gold, and Mark Harman. 2007. An empirical study of static program slice size. *ACM Trans. Softw. Eng. Methodol.* 16, 2 (April 2007), 8–es. doi:10.1145/1217295.1217297

[13] BlockSec. 2026. Ensuring a Secure and Seamless Web3 World. https://blocksec.com/.

[14] BNB Chain community. 2026. About the BEP Category. https://forum.bnbchain.org/t/about-the-bep-category/624.

[15] Vitalik Buterin, Sam Wilson, Ansgar Dietrichs, and lightclient. 2024. *EIP-7702: Set Code for EOAs.* https://eips.ethereum.org/EIPS/eip-7702 Ethereum Improvement Proposals.

[16] CaptPython. 2019. Design pattern Checks-Effects-Interactions Pattern. https://ethereum.stackexchange.com/questions/66456/design-pattern-checks-effects-interactions-pattern.

[17] CFI Team. 2026. Hard Forks. https://corporatefinanceinstitute.com/resources/cryptocurrency/hard-fork/.

[18] 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. doi:10.1145/3563328

[19] Che Kohler. 2022. What Is Bitcoin Block Time? https://thebitcoinmanual.com/articles/btc-block-time/.

[20] Chong Chen, Jianzhong Su, Jiachi Chen, Yanlin Wang, Tingting Bi, Jianxing Yu, Yanli Wang, Xingwei Lin, Ting Chen, and Zibin Zheng. 2025. When ChatGPT Meets Smart Contract Vulnerability Detection: How Far Are We? *ACM Trans. Softw. Eng. Methodol.* 34, 4, Article 100 (April 2025), 30 pages. doi:10.1145/3702973

[21] Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2025. Reasoning Runtime Behavior of a Program with LLM: How Far are We? . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE).* IEEE Computer Society, Los Alamitos, CA, USA, 1869–1881. doi:10.1109/ICSE55347.2025.00012

[22] 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. doi:10.1109/TSE.2020.2989002

[23] 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. doi:10.1109/SANER50967.2021.00050

[24] Noah Citron and Valeria Nikolaenko. 2024. Understanding Dencun, the biggest upgrade to Ethereum since The Merge. https://a16zcrypto.com/posts/article/understanding-dencun-upgrade-protodanksharding-surge-merge/. Accessed: 2026-05-28.

[25] code423n4. 2023. *Discrepancy in ECRECOVER Precompile when Using Delegatecall.* Code4rena. https://github.com/code-423n4/2023-10-zksync-findings/issues/175 GitHub Issue #175.

[26] code423n4. 2023. *tx.origin may be removed in future and its usage for contract check is not recommended.* Code4rena. https://github.com/code-423n4/2023-05-base-findings/issues/131 GitHub Issue #131.

[27] Code4rena. 2023. *M-19: Divergences in the Simulation of the EXTCODEHASH EVM Opcode.* Cyfrin Solodit. https://solodit.cyfrin.io/issues/m-19-divergences-in-the-simulation-of-the-extcodehash-evm-opcode-code4rena-zksync-zksync-git Security issue report for zkSync Era.

[28] Consensys. 2019. Stop Using Solidity's transfer() Now. https://consensys.io/diligence/blog/2019/09/stop-using-soliditys-transfer-now/.

[29] ConsenSys. 2026. Consensys - A complete suite of trusted products to build anything in web3. https://consensys.io/.

[30] Crytic. 2026. Rattle: EVM Binary Analysis Framework. https://github.com/crytic/rattle. Accessed: 2026-05-28.

[31] Crytic. 2026. SlithIR - Slither Wiki. https://github.com/crytic/slither/wiki/SlithIR. Accessed: 2026-05-28.

[32] Cyfrin. 2026. Solodit. https://solodit.cyfrin.io/.

[33] DefiLlama. 2026. DefiLlama EVM Chains. Retrieved from https://defillama.com/chains/EVM.

[34] Optimism Docs. 2026. Opcode Differences. https://docs.optimism.io/stack/differences#opcode-differences. Accessed: 2026-05-28.

[35] Ethereum.org. 2026. Gas and fees. https://ethereum.org/developers/docs/gas/.

[36] Etherscan. 2026. Etherscan Smart Contracts Audit and Security. https://etherscan.io/directory/Smart_Contracts/Smart_Contracts_Audit_And_Security.

[37] evm.storage. 2026. An Ethereum Virtual Machine Opcodes Interactive Reference. https://www.evm.codes/.
[38] Ethereum Stack Exchange. 2026. Ethereum Stack Exchange. https://ethereum.stackexchange.com/.
[39] 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.
[40] HarpOnLife. 2023. A Guide to Crafting Effective Prompts for Diverse Applications. https://community.openai.com/t/a-guide-to-crafting-effective-prompts-for-diverse-applications/493914. Accessed: 2026-05-28.
[41] Arturo Roura Heiko Fisch. 2024. Enforcers: Miscellaneous Minor Points. MetaMask DeleGator Security Audit via Solodit. https://solodit.cyfrin.io/issues/enforcers-miscellaneous-minor-points-consensys-none-metamask-delegator-markdown Accessed: 2026-05-28.
[42] 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. doi:10.1145/3597206
[43] Inspex. 2022. 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.
[44] Shray Jain. 2022. An Introduction to the Solana EVM. https://www.alchemy.com/overviews/solana-evm. Published on Alchemy, Accessed: 2026-05-28.
[45] 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. doi:10.1145/3560832.3563442
[46] 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. doi:10.1145/2976749.2978309
[47] 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. doi:10.1109/TSE.2021.3117966
[48] Martin Lundfall (@Mrchico). 2020. ERC-2612: Permit Extension for EIP-20 Signed Approvals. https://eips.ethereum.org/EIPS/eip-2612.
[49] 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.
[50] Offchain Labs. 2026. *Solidity Support: Differences from Solidity on Ethereum.* Arbitrum Documentation. https://docs.arbitrum.io/build-decentralized-apps/arbitrum-vs-ethereum/solidity-support#differences-from-solidity-on-ethereum
[51] OpenZeppelin. 2026. openzeppelin-contracts. https://github.com/OpenZeppelin/openzeppelin-contracts.
[52] Orderly Network. 2023. What is an EVM Compatible Chain? https://medium.com/@orderlynetwork/what-is-an-evm-compatible-chain-46a7825adc4d.
[53] 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. doi:10.1109/SANER50967.2021.00089
[54] Polkadot Documentation. 2026. Add Smart Contract Functionality. https://docs.polkadot.com/develop/parachains/customize-parachain/add-smart-contract-functionality/. Accessed: 2026-05-28.
[55] Polygon Technology. 2026. Processing L2 Blocks: The BLOCKHASH Opcode. https://docs.polygon.technology/zkEVM/architecture/proving-system/processing-l2-blocks/. Accessed: 2026-05-28.
[56] Quantstamp. 2026. *Will EIP-7702 Affect Your Code?* Quantstamp. https://quantstamp.com/blog/will-eip-7702-affect-your-code
[57] Chanchal Kumar Roy and James R Cordy. 2007. *A survey on software clone detection research.* Technical Report 2007-541. Queen's University, School of Computing. Technical Report.
[58] SlowMist. 2026. SlowMist - Focusing on Blockchain Ecosystem Security. https://www.slowmist.com/.
[59] Soliditydeveloper. 2022. What is ecrecover in Solidity? https://soliditydeveloper.com/ecrecover.
[60] Stack Overflow. 2026. Stack Overflow - Where Developers Learn, Share, & Build Careers. https://stackoverflow.com/.
[61] 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. doi:10.1145/3611643.3616270
[62] Tanish Gupta. 2023. EOAs vs Contracts: Understanding the Two Types of Ethereum Accounts. https://medium.com/@tanish_gupta/eoas-vs-contracts-understanding-the-two-types-of-ethereum-accounts-378f9402d0e8.
[63] Trail of Bits. 2026. Trail of Bits. https://www.trailofbits.com/.
[64] TRON. 2026. Decentralize The Web. https://tron.network/.
[65] 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. doi:10.1109/ICSE.2015.59
[66] tynes. 2024. *Interop: Address Aliasing*. Ethereum Optimism Specifications. https://github.com/ethereum-optimism/specs/issues/76 GitHub Issue #76.
[67] Uniswap. 2021. Router02. https://docs.uniswap.org/contracts/v2/reference/smart-contracts/router-02.
[68] VeChain. 2023. Contract address prediction. https://docs.vechain.org/core-concepts/evm-compatibility/test-coverage/contract-address-prediction.
[69] 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. doi:10.1109/IVS.2018.8500488
[70] Sally Junsong Wang, Kexin Pei, and Junfeng Yang. 2024. SmartInv: Multimodal Learning for Smart Contract Invariant Inference. In *2024 IEEE Symposium on Security and Privacy (SP)*. 2217–2235. doi:10.1109/SP54263.2024.00126
[71] 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. doi:10.1145/3643734
[72] 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 (June 2025), 23 pages. doi:10.1145/3728921
[73] 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. doi:10.1109/TSE.2024.3427321
[74] Wikipedia contributors. 2026. Binance — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Binance&oldid=1214652916.
[75] Wikipedia contributors. 2026. Ethereum — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Ethereum&oldid=1214478940.
[76] Wikipedia contributors. 2026. Polygon (blockchain) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Polygon_(blockchain)&oldid=1214411541.
[77] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.
[78] Jed R Wood and Larry E Wood. 2008. Card sorting: current practices and beyond. *Journal of Usability Studies* 4, 1 (2008), 1–6.
[79] Shuo Yang, Jiachi Chen, and Zibin Zheng. 2023. Definition and Detection of Defects in NFT Smart Contracts. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 373–384. doi:10.1145/3597926.3598063
[80] Zhemin Yang and Min Yang. 2012. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In *2012 Third World Congress on Software Engineering*. 101–104. doi:10.1109/WCSE.2012.26
[81] 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. doi:10.1145/3597503.3639131
[82] Zibin Zheng, Kaiwen Ning, Qingyuan Zhong, Jiachi Chen, Wenqing Chen, Lianghong Guo, Weicheng Wang, and Yanlin Wang. 2025. Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering* 30, 2 (2025), 50.
[83] 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. doi:10.1109/TSE.2019.2942301