

One Signature, Multiple Payments: Demystifying and Detecting Signature Replay Vulnerabilities in Smart Contracts

Zexu Wang
 Sun Yat-sen University
 Zhuhai, China
 Peng Cheng Laboratory
 Shenzhen, China
 wangzx97@mail2.sysu.edu.cn

Wenqing Chen*
 Sun Yat-sen University
 Zhuhai, China
 chenwq95@mail.sysu.edu.cn

Yuming Feng
 Peng Cheng Laboratory
 Shenzhen, China
 fengym@pcl.ac.cn

Jiachi Chen
 Sun Yat-sen University
 Zhuhai, China
 Zhejiang University
 Hangzhou, China
 chenjch86@mail.sysu.edu.cn

Kaiwen Ning
 Sun Yat-sen University
 Zhuhai, China
 Peng Cheng Laboratory
 Shenzhen, China
 ningkw@mail2.sysu.edu.cn

Yu Zhang
 Harbin Institute of Technology
 Harbin, China
 Peng Cheng Laboratory
 Shenzhen, China
 yuzhang@hit.edu.cn

Zibin Zheng
 Sun Yat-sen University, Guangdong
 Engineering Technology Research
 Center of Blockchain
 Zhuhai, China
 zhzibin@mail.sysu.edu.cn

Zewei Lin
 Sun Yat-sen University
 Zhuhai, China
 Peng Cheng Laboratory
 Shenzhen, China
 linzw3@mail2.sysu.edu.cn

Jianxing Yu
 Sun Yat-sen University
 Zhuhai, China
 yujx26@mail.sysu.edu.cn

Weizhe Zhang
 Harbin Institute of Technology
 Harbin, China
 Peng Cheng Laboratory
 Shenzhen, China
 wzzhang@hit.edu.cn

Abstract

Smart contracts have significantly advanced blockchain technology, and digital signatures are crucial for reliable verification of contract authority. Through signature verification, smart contracts can ensure that signers possess the required permissions, thus enhancing security and scalability. However, lacking checks on signature usage conditions can lead to repeated verifications, increasing the risk of permission abuse and threatening contract assets. We define this issue as the *Signature Replay Vulnerability* (SRV).

In this paper, we conducted the first empirical study to investigate the causes and characteristics of the SRVs. From 1,419 audit reports across 37 blockchain security companies, we identified 108

*Corresponding Author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE '26, Rio de Janeiro, Brazil

© 2026 ACM.

ACM ISBN 979-X-XXXX-XXXX-X/26/04
<https://doi.org/10.1145/XXXXXX.XXXXXX>

with detailed SRV descriptions and classified five types of SRVs. To detect these vulnerabilities automatically, we designed *LASiR*, which utilizes the general semantic understanding ability of *Large Language Models* (LLMs) to assist in the static taint analysis of the signature state and identify the signature reuse behavior. It also employs path reachability verification via symbolic execution to ensure effective and reliable detection. To evaluate the performance of *LASiR*, we conducted large-scale experiments on 15,383 contracts involving signature verification, selected from the initial dataset of 918,964 contracts across four blockchains: *Ethereum*, *Binance Smart Chain*, *Polygon*, and *Arbitrum*. The results indicate that SRVs are widespread, with affected contracts holding \$4.76 million in active assets. Among these, 19.63% of contracts that use signatures on *Ethereum* contain SRVs. Furthermore, manual verification demonstrates that *LASiR* achieves an *F1-score* of 87.90% for detection. Ablation studies and comparative experiments reveal that the semantic information provided by LLMs aids static taint analysis, significantly enhancing *LASiR*'s detection performance.

CCS Concepts

- Software and its engineering → Software testing and debugging

Keywords

Smart contracts, Signature replay vulnerability, LLM

ACM Reference Format:

Zexu Wang, Jiachi Chen, Zewei Lin, Wenqing Chen, Kaiwen Ning, Jianxing Yu, Yuming Feng, Yu Zhang, Weizhe Zhang, and Zibin Zheng. 2026. *One Signature, Multiple Payments: Demystifying and Detecting Signature Replay Vulnerabilities in Smart Contracts*. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/XXXXXX.XXXXXXX>

1 Introduction

Smart contracts, as programs running on the blockchain, directly impact the security of digital assets. Digital signatures are commonly used in contracts to verify that the signer has the required permissions. They employ encryption and hashing to securely verify transactions and data, enhancing both security and scalability. Digital signatures have been widely used in contract development for multi-signatures in asset management [2], maintaining information integrity between blockchains [43], and enabling functions like "checks" in digital assets [38], among others.

While digital signatures are widely used, the lack of checks on signature usage conditions (such as user identity requirements and the validity period) compromises the uniqueness of signature verification. This can result in the verification of a single signature multiple times, leading to *Signature Replay Vulnerabilities* (SRVs). In such scenarios, attackers reuse the same signature to pass multiple authorization checks for payments, illegally obtaining assets and compromising user trust and assets. A significant factor contributing to these vulnerabilities is developers' insufficient understanding of signature security practices. The research by Zhang et al. [63] reveals that 56.3% of contract developers face struggles to implement cryptographic practices, and 68.1% believe that the existing security tools need improvement. Many real-world developers are not cryptographic experts and lack experience in security development, leading to frequent signature replay attacks. For example, in July 2023, the *AzukiDAO* project lacked checks on signature usage, resulting in reused signatures and the \$69K asset loss [46]. Additionally, the signature verification status is often dispersed throughout the codebase and requires semantic analysis for validation, which attackers can exploit if there are errors or omissions in the verification process. For example, due to a developer error, the *branchMask* function in the *Polygon Plasma Bridge* generated the same signature for different *branch masks*, allowing the signature to be validated 223 times for burn transactions. This malleability of the signature enabled an attacker to steal \$22.3 million [30].

Signature Replay Vulnerabilities present new challenges for contract security. First, there is a lack of systematic research, making it difficult to summarize characteristics and design detection rules. Second, analyzing dangerous signature verification patterns in complex contracts requires a comprehensive understanding of program semantics and contract intent, which is highly challenging.

To address these challenges, we conducted the first empirical study to summarize the causes and definitions of SRVs. Reflecting real developer issues, we manually examined 1,419 open-source contract audit reports from 37 security companies and identified

108 reports related to the reuse of signatures. Using the Open Card Sorting method [50], we classified dangerous patterns leading to signature reuse issues during verification. Finally, we identified five types of SRVs: *Cross-chain Replay Attack* (X-CRA), *Cross-project Replay Attack* (X-PRA), *Contract Account Signature Replay* (CASR), *Signature State Management Issue* (SSMI), and *Signature Malleability Attack* (SMA) (see Subsection 3.4 for details).

To effectively detect SRVs, we designed a tool named *LASiR*, which utilizes *Large Language Models* (LLMs) [31] to understand contract semantics, combining static taint analysis and symbolic execution to enhance detection reliability. It inputs smart contract source code and outputs detection results in three phases: *Slicing with LLM Analysis*, *Inspection of Signature Verification*, and *Path Reachability Verification*. In *Phase 1*, *LASiR* utilizes LLMs to identify variables related to signature states and analyzes their dependencies to perform program slicing. In *Phase 2*, *LASiR* leverages LLMs to analyze the semantic information within the slices related to signature verification. It identifies sanitized variables to assist in taint analysis status checks, detects hazardous signature verification patterns, and generates *Warnings*. In *Phase 3*, *LASiR* requires LLMs to review *Warnings* and provide function sequences to guide symbolic execution for path reachability verification. *LASiR* leverages LLMs' general understanding ability to assist in static taint analysis, ensuring detection accuracy and reliability.

We conducted three experiments to evaluate *LASiR*'s detection performance for SRVs. First, we crawled 918,964 contract source codes from *Ethereum* [24], *Binance Smart Chain* [58], *Polygon* [59], and *Arbitrum* [3] to analyze performance on large-scale datasets. By analyzing the contract's AST file, we screened 15,383 contracts related to signature verification (*DB1*). Experiments revealed that SRVs are widespread, with affected contracts holding \$4.76 million in active assets. Among these, 19.63% of contracts that use signatures on *Ethereum* contain SRVs. The average detection time is approximately 40 seconds, with a total LLM API cost of around \$15, demonstrating *LASiR*'s efficiency and low cost for detection. To further analyze *LASiR*'s effectiveness, we randomly selected 500 contracts from *DB1* for manual analysis, identifying 72 positive and 428 negative cases (*DB2*). *LASiR* achieved a *Precision* of 82.14%, a *Recall* of 95.83%, and an *F1-score* of 88.46%, outperforming the compared general-purpose tools. Additionally, ablation experiments analyzing the impact of LLM on *LASiR*'s performance showed significant improvements: *Precision* increased from 4.40% to 82.14%, *Recall* from 26.39% to 95.83%, and *F1-score* from 7.54% to 88.46%. The information provided by LLM through contract context analysis is essential for static taint analysis, enhancing accuracy and efficiency.

The main contributions of this work are as follows:

- This study conducted the first empirical analysis of SRVs in smart contracts. We manually examine real-world security audit reports, define five types of SRVs, and provide explanatory examples.
- We designed *LASiR*, leveraging LLM's semantic understanding to assist in static taint analysis of the signature state, achieving efficient detection of SRVs.
- We provide a dataset of real-world SRVs from 918,964 contracts across four blockchains. This dataset identifies 1,739 contracts with SRVs holding \$4.76 million in assets, which

can further aid research in vulnerability repair and automated exploitation efforts.

- We have open-source LASiR's tool code, experimental data, and empirical research data at <https://anonymous.4open-science/r/LASiR-B207>.

2 Background

2.1 Signature Verification of Smart Contracts

The digital signature is an effective method to verify the identity of the signer [13]. For example, contracts using the *Elliptic Curve Digital Signature Algorithm* (ECDSA) [57] allow the signer to sign a message with the private key, while anyone can verify the signature with the public key, ensuring the authenticity and integrity of the signed message. Ethereum deploys the verifying signature program in *ecrecover()* (precompiled contract at address *0x1*) for on-chain verification [12]. *ecrecover()* inputs the message hash (*_messageHash*) and signature parameters (*v, r, s*), recovers the signer's address. The message hash (*_messageHash*) is derived from the Keccak-256 hash [44] operation on the signed message. The signature parameters (*v, r, s*) result from the signer signing the *_messageHash*, following the *Secp256k1* cryptographic algorithms [22]. Developers can directly call *ecrecover()* in contracts to recover the signer's address, enabling identity verification and token delegation authorization (similar to "checks" in digital assets) [32], among others.

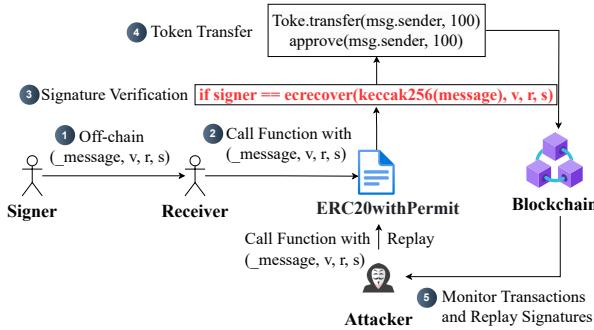


Figure 1: Signature Replay Attack Process.

Figure 1 illustrates the signature replay attack process. The *ERC20withPermit* contract, derived from the ERC-20 token standard [14], allows users to authorize other accounts to transfer tokens without additional transactions. The process is outlined as follows: ① The signer signs the *_message*, generates the signature information (*v, r, s*), and notifies the receiver through an off-chain channel. ② The receiver submits the message (*_message*) and signature information (*v, r, s*) as input parameters to the *ERC20withPermit* contract, requesting a token transfer. ③ The contract uses the *ecrecover()* function to recover the address from the signature and message, checking if the recovered address matches the signer's address. ④ If the verification is successful, the contract authorizes the *msg.sender* (receiver) to execute the transfer operation with a limit of 100 tokens. However, signature verification lacks identity checks on the *msg.sender*, allowing anyone to reuse the signature and posing significant threats to asset security. As shown in Figure 1, ⑤ an attacker can monitor blockchain transactions to obtain the message (*_message*) and signature information (*v, r, s*) and reuse

them to submit a transfer request to the *ERC20withPermit* contract. Due to the lack of identity checks on the *msg.sender*, the contract permits the attacker (*msg.sender*) to transfer 100 tokens.

2.2 Static Taint Analysis

Static taint analysis tracks data flow through predefined patterns to detect security vulnerabilities. The process comprises three main steps: identifying taint sources, analyzing taint propagation, and identifying and checking sinks. In smart contracts, taint sources include user inputs (e.g., *msg.value*, *msg.data*, *msg.sender*), external blockchain attributes (e.g., *block.timestamp*, *block.chainid*), and return values from external contract calls (e.g., *call*, *delegatecall*). Sinks need to be custom-defined based on specific detection tasks and expert rules. Taint propagation analysis involves data flow and control flow analysis. Data flow analysis tracks the flow of tainted data through assignments and function calls, while control flow analysis examines whether tainted data reaches sensitive operations. This method effectively analyzes dependencies within smart contracts, making it a robust approach for detecting vulnerabilities. However, the accuracy of detection is highly dependent on the precise extraction of contract semantics. Many methods use fixed pattern matching for semantic feature extraction, resulting in challenges such as low automation and insufficient semantic analysis capabilities. Patterns heavily rely on manual experience, further constraining detection capabilities in complex scenarios.

3 Signature Replay Vulnerability Definition

In this section, we conduct an empirical study on real-world security audit reports regarding signature reuse to define and classify common *Signature Replay Vulnerabilities* (SRVs) in smart contracts.

3.1 Data Collection

To comprehensively analyze real-world issues related to signature reuse, we collected open-source security audit reports from various security teams. Specifically, we accessed the public URLs (official websites, X (Twitter), and GitHub) of 81 smart contract security teams listed by Etherscan [23]. Of these, 37 teams had public audit reports, including *BlockSec* [52], *Trail of Bits* [39], and *SlowMist* [48]. Additionally, we gathered vulnerability audit reports from bug bounties publicly available on *Solodit* [17]. In total, we manually collected 1,419 security audit reports.

3.2 Data Pre-processing

To filter security reports related to SRVs, we combined keyword filtering with manual checks. Initially, we selected commonly used terms in signature verification as keywords, including "*ecrecover()*", "*signature*", and "*replay attack*", with "*ecrecover()*" specifically chosen for its significance in signature verification. Automated keyword filtering identified 557 reports (467 with "*signature*", 28 with "*ecrecover()*", and 62 with "*replay attack*"), each containing at least one keyword. However, the multiple meanings of keywords can easily lead to misidentifications, as some reports may contain these keywords but are unrelated to signature replay. For example, the report [37] highlights an issue with the "*incorrect function signature*", which was selected due to the "*signature*" keyword matching but is unrelated to SRVs. Therefore, manual checking is necessary to

eliminate these irrelevant reports. Finally, through manual filtering, we obtained 108 security audit reports related to SRVs.

3.3 Data Analysis

To classify SRVs, we used the Open Card Sorting method [50], widely employed for problem discovery and definition in software engineering [7, 34, 65]. We created a card for each audit report, detailing the *Title*, *Descriptions*, *Root Causes*, and *Recommendations*. For example, Figure 2 presents the card content of the report [53], highlighting key sections such as the *Title* and *Root Causes*, which directly indicate the lack of checks on the *signatureClaimed[_signature]* status, leading to signature reuse. The *Descriptions* provides relevant case analysis, and the *Recommendation* section outlines the mitigation measures. These professionally-audited reports offer structured content for quick vulnerability analysis.

Issue: Signature Status Verification Error Leading to Replay Attack	
function claim(...) { ... bytes32 messageHash=keccak256(abi.encodePacked(msg.sender,_claimAmount)); require(signatureManager == recover(messageHash,_signature),"invalid signature"); signatureClaimed[_signature] = true //transfer token _transfer(address(this),msg.sender,_claimAmount);	Descriptions
}	
Root Causes: Although the <i>signatureClaimed[_signature]</i> records whether a signature has been used, there is no check on its usage status during verification.	
Recommendation: Add a check for <i>signatureClaimed[_signature]</i> right before transferring the token. This can be achieved by adding the following: require(!signatureClaimed[_signature], "Signature already used!!!");	

Figure 2: Example of the Audit Report Card.

To accurately and objectively characterize real-world vulnerabilities, we adopt a classification framework grounded in three key dimensions: (i) commonly accepted industry terminology, (ii) prevalent developer challenges, and (iii) specific exploitation conditions. Domain experts with extensive experience in smart contract security ensure alignment with the language used in professional audit reports. For instance, *signature malleability* is categorized based on its cryptographic underpinnings in the Elliptic Curve Digital Signature Algorithm (ECDSA). We further emphasize that various signature-related vulnerabilities (SRVs) originate from distinct root causes and triggering mechanisms. For example, X-CRA and X-PRA stem from separate misconfigurations in blockchain identity verification and project address validation, respectively. Moreover, we observe that vulnerabilities such as *Signature State Management Issues* (SSMI) frequently arise due to developers' limited security awareness and flawed design logic, particularly in managing complex digital signature states within smart contracts. For a comprehensive explanation of the classification criteria and illustrative examples, please refer to Part I of Appendix A [33].

We invited two experienced smart contract security researchers to classify the vulnerabilities without predefined categories.

1 First Round. The researchers randomly selected 40% of the vulnerability cards for initial classification. They examined each card by reading the *Title* and *Description* to understand the vulnerability, analyzing the problematic code to identify the *Root Cause*, and reviewing the *Recommendations* to infer related types. If a

vulnerability did not fit any existing category, they evaluated its representativeness and recurrence before proposing a new category. This round resulted in five preliminary types of signature replay vulnerabilities (SRVs): *Cross-Chain Replay Attacks*, *Cross-Project Replay Attacks*, *Signature State Management Issues*, *Signature Malleability Attacks*, and *Front-Running Replay Attacks*. Since all cards were sourced from real-world security audit reports with clearly defined content, no cards were excluded as irrelevant.

2 Second Round. The researchers independently classified the remaining 60% of the cards. During this process, they identified an additional category: *Contract Account Signature Replay*.

3 Reconciliation. The researchers then compared their labeling results. The main disagreements involved four cards, primarily concerning the distinction between *Front-Running Replay* and *Contract Account Signature Replay*. After discussion, they agreed that the latter exhibited distinct exploitation characteristics and had broad real-world impact, warranting its recognition as a standalone category. Conversely, the definition of *Front-Running Replay* was deemed overly narrow, and its instances could be subsumed under other categories. As a result, this category was removed. Ultimately, five SRV types were finalized. During the classification of all 108 cards, the two researchers disagreed on only 4 cards, yielding a disagreement rate of $D = \frac{N_{\text{disagree}}}{N_{\text{total}}} = \frac{4}{108} \approx 3.7\%$, which reflects the clarity and consistency of the audit report content. The complete labeling results, along with the classification criteria and examples (detailed in Part II of Appendix A), are available in repository [33].

3.4 Definition of Signature Replay Vulnerability

In this subsection, we define five types of SRVs, as shown in Figure 3 with their corresponding *Definitions* and *IDs*. Each type is detailed and explained with *Code Examples* from real-world audit reports.

ID	Vulnerability Type	Definition
X-CRA	Cross-chain Replay Attack	Signatures can be replayed across different chains due to the lack of Blockchain ID verification.
X-PRA	Cross-project Replay Attack	Signatures can be replayed between contracts with the same code from different projects due to the lack of contract address verification.
CASR	Contract Account Signature Replay	Contract account signatures can be reused across multiple contract accounts of one EOA due to the lack of contract account address verification.
SSMI	Signature State Management Issue	Signatures can be reused due to incorrect signature management or updates to the signature usage state.
SMA	Signature Malleability Attack	ECDSA-generated signatures can be manipulated to generate different signatures for the same message, potentially leading to attacks.

Figure 3: Definitions of Five Types of SRVs.

(1) Cross-chain Replay Attack (X-CRA). The *Blockchain ID* is a unique identifier for distinguishing one blockchain from another [60]. During signature verification, checking the *Blockchain ID* in the signature message can restrict verification to a specific blockchain [6]. The absence of *Blockchain ID* checks in the signature message can lead to the same signature being reused across multiple blockchains, resulting in X-CRA.

Code Example: Figure 4 illustrates the X-CRA identified by auditors in the *Bconomy* project. The issue arises from the *getHash()* function, not including *block.chainid* in the signature message, invalidating blockchain-specific restrictions of the signature verification. Auditors noted that deploying this code on two blockchains

would allow the same signature to be verified on both. They recommended adhering to the *EIP-4337 standard* [6], which includes adding `block.chainid` to the signature message to prevent X-CRA.

```

1 function getHash(UserOperation userOp) public {
2     keccak256(abi.encode(
3         block.chainid // @audit add chain id
4         ...
5     ), userOp.getSender(), userOp.nonce));
6 }
```

Figure 4: X-CRA in Bconomy Project [9].

(2) Cross-project Replay Attack (X-PRA). The contract address is a critical credential for distinguishing between projects deployed with the same code. Without contract address checks during the signature verification, the same signature can be reused by different projects. For instance, an attacker could reuse a valid signature from one project to execute unauthorized actions on another project.

```

1 function _checkSig(...) public {
2     bytes32 messageDigest = keccak256(...,
3         address(this)); // @audit add contract address
4     ecrecover(messageDigest, v, r, s);
5     ...
6 }
```

Figure 5: X-PRA in Hermez Project [28].

Code Example: Figure 5 illustrates the X-PRA discovered by auditors in the *Hermez* project. The `_checkSig()` function does not include the project address (`address(this)`) when hashing the signature message. This omission allows the same signature to be verified across different *Hermez* forked projects. For example, an attacker could reuse a signature verified by the *Hermez* project in a forked instance (*Hermez_forked*) on the same blockchain to steal funds. To prevent this, it is recommended to include the project contract address in the signature message, ensuring that signatures cannot be reused in different projects.

(3) Contract Account Signature Replay (CASR). *Contract Account Signature*, where the contract itself signs instead of an *Externally Owned Account* (EOA) using its private key [10, 27]. This process follows the *EIP-1271 standard* [26], which allows an EOA to create multiple contract accounts and sign securely. However, if contract account address checks are omitted during signature verification, the same signature could be validated by different contract accounts, resulting in CASR.

```

1 function execScheduled(Identity identity, bytes32
2     accHash, uint nonce, ... calldata txns) external {
3     bytes32 hash = keccak256(..., address(identity),
4         ...
5     );
6     require(scheduled[hash] != 0 && ...);
```

Figure 6: CASR in AdEx protocol [8].

Code Example: Figure 6 presents an example of CASR in the *AdEx protocol*, which utilizes the *EIP-1271 standard* for *QuickAccount* [29] to manage multiple identities (contract accounts) and verify contract signatures. However, as shown in line 2 of Figure 6, the signature message lacks the identity address, allowing it to be verified by multiple identities (contract accounts) and leading to asset loss. Auditors recommend incorporating the identity address into the signature message to prevent CASR.

(4) Signature State Management Issue (SSMI). To ensure effective signature state management, *EIP-712* [5] introduces the `domainSeparator`, which structurally organizes signature information such as `timestamps` and `nonces`, facilitating the verification and management of signature states. However, due to insufficient security awareness among developers, custom flawed signature management often results in disordered signature states. These vulnerabilities in signature state management can be exploited by attackers to perform replay attacks.

```

1 function recoverSignature(...) returns (address) {
2     ...
3     SignedData memory payload = SignedData({
4         transactionId: transactionId, ...
5     });
6     return ECDSA.recover(ECDSA.toEthSignedMessageHash(
7         keccak256(abi.encode(payload))), signature);
8 }
```

Figure 7: SSMI in Connex NXTP [11]

Code Example: In Figure 7, the `recoverSignature()` function generates the custom signature message to verify the signer's identity. However, the lack of a mechanism to track signature usage results in disordered signature states, leading to SSMI. Auditors recommend strictly adhering to the *EIP-712* during signature verification [11], including the use of a nonce to track and prevent signature reuse.

(5) Signature Malleability Attack (SMA). The *Elliptic Curve Digital Signature Algorithm* (ECDSA) employed by *Ethereum* is vulnerable to signature malleability attacks [1]. In such attacks, an attacker can modify specific parts of a signature without access to the private key, creating a new valid signature corresponding to the original signature message. To avoid such attacks, it is essential to enforce restrictions on the variables `v` and `s` when using the `ecrecover(hash, v, r, s)` function. However, due to insufficient secure development expertise, developers often directly use `ecrecover(hash, v, r, s)` for verification without implementing these checks.

```

1 function permit(owner, spender, amount, v, r, s) {
2     bytes32 permitDataDigest = keccak256(abi.encode(
3         PERMIT_TYPEHASH, owner, spender));
4     bytes32 digest = keccak256(abi.encodePacked("\x19\x01", DOMAIN_SEPARATOR(), permitDataDigest));
5     require(owner == ecrecover(digest, v, r, s));
6     allowances[owner][spender] = amount;
7 }
```

Figure 8: SMA in Interest Protocol [42].

Code Example: Figure 8 shows the SMA in the *Interest Protocol* [42], where the `permit()` function directly calls `ecrecover()` for signature verification, exposing the protocol to replay attacks. Auditors recommend the following measures to mitigate this risk: 1) Ensure the `s` value falls within the range $0 < s < \text{secp256k1n} \div 2 + 1$ (the lower half of the range). 2) Restrict the `v` value to 27 or 28. Furthermore, adopting secure contract libraries, such as version 4.7.3 or later of *OpenZeppelin's* ECDSA library [41], ensures unique signature verification and prevents malleability attacks. These measures can effectively reduce the risk of signature malleability.

4 Methodology

In this section, we introduce LASiR's methodology, utilizing LLMs to understand contract semantics, combining static taint analysis and symbolic execution to enhance detection reliability.

4.1 Overview

Figure 9 shows an overview of LASiR, which inputs smart contract source codes and outputs the detection results. The detection process consists of three phases: *Slicing with LLM Analysis*, *Inspection of Signature Verification*, and *Path Reachability Verification*.

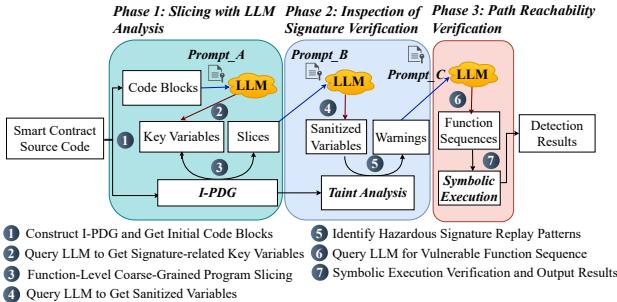


Figure 9: Overview of LASiR.

Phase 1: Slicing with LLM Analysis. To mitigate the impact of LLM input length limitations, we construct the *Inter-contract Program Dependency Graph* (I-PDG) [56] to comprehensively analyze dependencies and slice the code related to signature verification. The I-PDG represents global state dependencies, allowing for quick identification of dependencies between variables and statements. LASiR integrates the *Initial Code Block* that calls `ecrecover()` with *Prompt_A* to query the LLM for key variables related to signature verification. Based on the dependencies of these key variables from the I-PDG, it slices the code that executes signature verification.

Phase 2: Inspection of Signature Verification. LASiR utilizes the general understanding capabilities of LLMs for automated state inspection, enhancing the precision of the static taint analysis. LASiR uses *Prompt_B* to query the LLM for sanitized variables about the signature from sliced code. During taint analysis, it examines the dependencies of these variables to ensure sources are not contaminated by the time they reach sinks. By combining various *domain-specific patterns* (details for Section 4.5), it identifies risky behaviors and outputs relevant function names as *Warnings*.

Phase 3: Path Reachability Verification. To enhance detection reliability, LASiR employs self-validation through symbolic execution to verify path reachability. LASiR uses *Prompt_C* to instruct the LLM to understand the semantic context of functions from *Warnings* and generate the sequence of functions containing risky logic or operations. Subsequently, symbolic execution is employed to explore execution paths derived from different function sequences, proving their reachability, and outputting detection results.

4.2 Slicing with LLM Analysis

4.2.1 Step 1: Construct I-PDG and Get Initial Code Blocks. LASiR takes smart contract source code as input, compiles it into the *Abstract Syntax Tree* (AST), and subsequently analyzes the AST to generate the *Inter-contract Program Dependency Graph* (I-PDG) [56]. The I-PDG integrates global control flow and inter-contract calls from the *Inter-contract Control Flow Graph* (I-CFG) [36], supplements data dependencies, and establishes global program dependencies. The nodes in the graph are derived from AST statements,

while the edges represent data and control dependencies to model the program's structure. Specifically, as shown in Figure 10, the numbers in circles (nodes) correspond to line numbers in Figure 8, with red, blue, and green lines representing data dependencies, control dependencies, and inter-contract calls, respectively. Figure 10 illustrates the slicing analysis with LLMs for the *Interest Protocol*. LASiR collects all nodes within functions that contain the `ecrecover()` call as *Initial Code Blocks* and further analyzes them using LLMs. We define *Initial Code Blocks* as the set of nodes that include the `ecrecover()` invocation along with all its dependent nodes. As shown in Figure 10, this includes not only lines 1–6 in Figure 8, but also the `PERMIT_TYPEHASH` state variable and the call to the `DOMAIN_SEPARATOR()` function. All nodes related to the dependencies of `DOMAIN_SEPARATOR()` are also included in the slice.

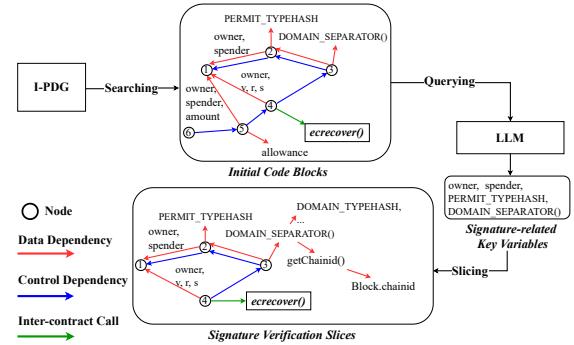


Figure 10: Slicing with LLM Analysis for *Interest Protocol* [42].

4.2.2 Step 2: Query LLM to Get Signature-related Key Variables. Static analysis often struggles to extract task-specific dependencies due to limited semantic understanding, impacting detection accuracy. To address this, we designed *Prompt_A* to guide the LLM in analyzing signature variables based on contract semantics and intent. These variables and their dependencies help static analysis focus on the signature status, thereby enhancing slicing accuracy.

We designed *Prompt_A* based on common practices [45] and *Tier of Thought* (ToT) design [55]. The analysis task is divided into three tiers for complex tasks, using outputs from previous tiers to generate responses for more challenging tasks, ensuring the reliability of the LLM's outputs. For structured output, the LLM returns results in JSON format only in the final round to maintain its thought process. Specifically, *Prompt_A* guides the LLM to simulate a smart contract security auditor's workflow, aiding in accurately identifying and extracting key variables for signature verification through role-playing and structured analysis.

As shown in Figure 11, *Prompt_A* consists of four parts: *Role Playing*, *Task Definition*, *Step-by-Step Analysis*, and *Output Format*. In *Role Playing*, we define the LLM as a smart contract security auditor skilled in identifying and mitigating vulnerabilities, activating its vulnerability analysis capabilities. The *Task Definition* clearly states the task: extract variable names related to signature verification (`ecrecover()`) from the `%Initial Code Blocks%`. The *Step-by-Step Analysis* breaks the task into three tiers: 1) Determine if the code implements the signature verification. 2) If the code does implement signature verification, extract all

state variables involved. 3) Filter out variables that affect signature verification. The *Output Format* specifies that the output should be a *JSON* object containing the required information. Since static analysis often retains irrelevant variables due to a lack of semantic context, it compromises accuracy. By leveraging the LLM, we can effectively obtain key variables related to signatures. As shown in Figure 10, when analyzing *Signature-related Key Variables*, the LLM, with its semantic analysis, filters out irrelevant variables such as *allowance* in line 5 of Figure 8 (whereas static analysis considers it due to control dependencies), thereby improving the accuracy.

Role Playing: As a smart contract security auditor proficient in vulnerability identification and mitigation, please complete the following task based on the provided instructions.
Task Definition: Extract variable names related to the signature verification (e.g., <code>recover</code>) execution from the <code>{%Initial Code Blocks%}</code> .
Step-by-Step Analysis: The prompt outlines a structured three-tiers method for analysis:
<ol style="list-style-type: none"> 1. Identify whether <code>{%Initial Code Blocks%}</code> execute the signature verification process. 2. If the code implements signature verification, extract all state variables involved in the process. 3. Filter out the state variables related to signature verification.
Output Format: The expected output format is specified as a JSON object with three key-value pairs:
<ul style="list-style-type: none"> • <code>is_signature_verification_implemented</code>: "yes" / "no". • <code>all_involved_variables</code>: List of all variables involved. • <code>variables_related_to_signature_verification</code>: List of variables related to signature verification.

Figure 11: *Prompt_A* Template Design.

4.2.3 Step 3: Function-Level Coarse-Grained Program Slicing. To mitigate the limitations of LLM on input text length while maintaining state completeness, *LASiR* employs function-level slicing based on program dependencies. It uses *Signature-related Key Variables* from *Step 2* and their dependencies to slice the code related to signature verification. Specifically, *LASiR* treats each function as a complete unit rather than analyzing individual statements. If any statement within a function depends on *Signature-related Key Variables*, the entire function is included. As shown in the *Signature Verification Slices* in Figure 10, key variable *DOMAIN_SEPARATOR()* relies on *Block.chainid* from the *getChainid()* function, thus the *getChainid()* function is included in the slice. *LASiR* uses a coarse-grained, function-level program slicing approach [4] to avoid the fragmentation of functions and compromise of information integrity that often occurs with fine-grained slicing. This approach ensures that all relevant statements are preserved, enhancing the semantic understanding and contextual integrity of LLM analysis.

4.3 Inspection of Signature Verification

4.3.1 Step 4: Query LLM to Get Sanitized Variables. In static taint analysis, taint data is sanitized into variables without sensitive data. However, due to the complexity and variability of sanitization processes, existing static analyses heavily rely on fixed rules or manual inspection, limiting their effectiveness. To automate variable sanitization analysis, we leverage the LLM's understanding of contract semantics, transforming the task into natural language instructions through prompt design and identifying sanitized variables.

To guide the LLM in identifying sanitized variables effectively, we designed *Prompt_B*. As shown in Figure 12, the prompt outlines the LLM's task to analyze sanitized variables related to `%sanitized_variable_type%` from the `{%provided_code%`. The analysis is structured into three tiers: 1) search for all variables checked

in signature verification and explain if none are found. 2) identify key variables based on their dependencies with the `%sanitized_variable_identification_rules%`, if signature verification is present. 3) If variables from tier 2 exist, use the specified `%sanitization_methods%` to determine which variables relevant to `%signature_replay_type%` are sanitized during verification. The table below categorizes the identification rules and sanitization methods for five types of SRVs. `%signature_replay_type%`, `%sanitized_variable_identification_rules%`, and `%sanitization_methods%` clarify the SRVs types, identification rules, and sanitization methods. This table guides the LLM in identifying sanitized variables and improving accuracy through targeted sanitization processes.

Role Playing: Assume the role of a smart contract security auditor with expertise in identifying and mitigating vulnerabilities.																		
Task Definition: Your task is to analyze the signature verification based on <code>{%provided_code%</code> to determine if there are sanitized variables related to <code>{%signature_replay_type%</code> .																		
Step-by-Step Analysis: This prompt outlines a structured, three-tiers method for your analysis:																		
<ol style="list-style-type: none"> 1. Determine whether the <code>{%provided_code%</code> performs signature verification, and if so, identify all variables checked during the process. If no variables are inspected, provide an explanation. 2. If the code implements signature verification, filter out the checked variables based on their dependencies with the <code>{%sanitized_variable_identification_rules%</code>. 3. If there are variables from step 2, use the specified <code>{%sanitization_methods%</code> to determine which variables relevant to <code>{%signature_replay_type%</code> are sanitized during verification. 																		
Output Format: The expected output format is a JSON object with three key-value pairs:																		
<ul style="list-style-type: none"> • <code>is_signature_verification_implemented</code>: "yes" / "no". • <code>all_checked_variables</code>: List of all variables checked. • <code>sanitized_variables_related_to_{%signature_replay_type%}</code>: List of sanitized variables during signature verification. 																		
<table border="1"> <thead> <tr> <th><code>signature_replay_type</code></th> <th><code>sanitized_variable_id</code></th> <th><code>sanitization_methods</code></th> </tr> </thead> <tbody> <tr> <td>X-CRA</td> <td>Blockchain ID</td> <td>Verify if immutable or properly checked</td> </tr> <tr> <td>X-PRA</td> <td>address(this)</td> <td>Ensure verification process checks address(this)</td> </tr> <tr> <td>CASR</td> <td>msg.sender</td> <td>Validate sender's address-related variables</td> </tr> <tr> <td>SMA</td> <td>v, r, s</td> <td>Properly validate to prevent malleability attack</td> </tr> <tr> <td>SSMI</td> <td>nonce, signature usage records</td> <td>Check for nonce or usage records, ensure restrictions based on time (e.g., block.timestamp)</td> </tr> </tbody> </table>	<code>signature_replay_type</code>	<code>sanitized_variable_id</code>	<code>sanitization_methods</code>	X-CRA	Blockchain ID	Verify if immutable or properly checked	X-PRA	address(this)	Ensure verification process checks address(this)	CASR	msg.sender	Validate sender's address-related variables	SMA	v, r, s	Properly validate to prevent malleability attack	SSMI	nonce, signature usage records	Check for nonce or usage records, ensure restrictions based on time (e.g., block.timestamp)
<code>signature_replay_type</code>	<code>sanitized_variable_id</code>	<code>sanitization_methods</code>																
X-CRA	Blockchain ID	Verify if immutable or properly checked																
X-PRA	address(this)	Ensure verification process checks address(this)																
CASR	msg.sender	Validate sender's address-related variables																
SMA	v, r, s	Properly validate to prevent malleability attack																
SSMI	nonce, signature usage records	Check for nonce or usage records, ensure restrictions based on time (e.g., block.timestamp)																

Figure 12: *Prompt_B* Template Design.

4.3.2 Step 5: Identify Hazardous Signature Replay Patterns. The LLM-identified sanitized variable information from *Step 4* lacks contract execution context and cannot be used directly for static taint analysis. To address this, *LASiR* searches for related code blocks based on the program dependencies of the sanitized variables. By combining these related code blocks with *domain-specific patterns*, it identifies risky signature patterns. Specifically, *LASiR* iterates through all sanitized variables to find their definition nodes and performs a *depth-first search* (DFS) of the entire I-PDG starting from these nodes. When dependencies on sanitized variables are found, the node is saved in *Warning_nodes*, and all successor nodes are recursively traversed. This generates a set of nodes that contain dependencies on sanitized variables (*Warning_nodes*), providing context for their operations. By combining these with *domain-specific patterns* based on different SRVs analysis strategies (see subsection 4.5), *LASiR* identifies hazardous signature patterns and generates the corresponding function names as *Warnings*.

4.4 Path Reachability Verification

4.4.1 Step 6: Query LLM for Vulnerable Function Sequence. To ensure reliable results, *LASiR* uses LLM's general understanding to review *Warnings* from static taint analysis in *Step 5*. *LASiR* uses *Prompt_C* to guide the LLM in checking and correcting these *Warnings*. If the *Warnings* are confirmed, the LLM provides a function

sequence related to the execution of the vulnerability. This sequence, which comprises multiple functions in a specific order, is generated by LLM reasoning. This information accelerates the traversal of the symbolic execution path, enhancing the efficiency of the analysis.

Role Playing: Assume the role of a smart contract security auditor with expertise in identifying and mitigating vulnerabilities.

Task Definition: (%Warnings%) represents critical functions in signature replay attacks. Your task is to analyze the business process related to function calls involving (%Warnings%) within (%provided_code%).

Step-by-Step Analysis: This prompt outlines a structured, three-tiers method for your analysis:

- Analyze the signature verification process in (%Warnings%) to determine the potential for signature reuse.
- If signature reuse is possible, output all sub-business flows, ensuring they include the function where (%Warnings%) is involved.
- The output business flows should only list functions within the contract itself, ignoring calls to other contracts or interfaces, as well as events.

Output Format: Provide a single result in JSON format with the structure: {(%Warnings%): [function1, function2, function3,...]}. The functions must be the function signature format.

Figure 13: *Prompt_C* Template Design.

Figure 13 shows *Prompt_C* template. In the task definition, %Warnings% is declared to be associated with signature replay attacks. The LLM's task is to analyze the business process related to function calls involving %Warnings% within %provided_code%. A structured three-tier analysis method guides the LLM in reasoning about the vulnerability function sequence, 1) analyzing the signature verification process in %Warnings% to determine the potential for signature reuse. 2) If signature reuse is possible, it should output all sub-business flows, ensuring %Warnings% is involved. 3) Output business flows should only list functions within the contract itself, ignoring calls to other contracts or interfaces, as well as events. The output format specifies that the results should be in JSON format with the structure: %Warnings%: [function1, function2, function3,...]. This JSON information contains the function sequence related to signature reuse. Utilizing the LLM's semantic understanding for pruning helps improve symbolic execution search efficiency.

4.4.2 Step 7: Symbolic Execution Verification and Output Results.

To avoid static taint analysis causing permissions to be ignored or extracted incorrectly, and to mitigate misunderstandings by the LLM that affect reliability, LASiR employs symbolic execution to verify path reachability. The function sequence from *Step 6* guides the symbolic executor to traverse the CFG along different paths and collect permission-related path constraints. These constraints are then verified using SMT-based satisfiability checks to confirm path reachability. If the path constraints are solvable, indicating that the relevant permission checks can be passed, the vulnerability is proven to exist. This approach avoids blind searches and mitigates path explosions. Combined with the *Warnings* identified by taint analysis, LASiR verifies path reachability through symbolic execution and outputs the results.

LASiR identifies feasible execution paths leading to *ecrecover()* based on CFG, and extracts both explicit and implicit control dependency conditions along these paths (e.g., *require(...)*, *if (msg.sender == owner)*). It then symbolically encodes the relevant variables by incorporating data dependencies, ultimately constructing complete path constraint expressions. Taking the *permit()* function illustrated in Figure 14 as an example, the symbolic path includes signature digest construction (*digest = keccak256(...)*), signature recovery (*s = ecrecover(...)*),

conditional checks (*require(owner == s)*), and state updates (*allowances[owner][spender] = amount*). The symbolic executor interprets the semantics of operations along the path and, using a symbolic memory model tailored to EVM instructions, generates corresponding path constraints. LASiR then submits the combined path constraints to an SMT solver (e.g., Z3) to check for satisfiability. If the result is SAT, it indicates the existence of input parameters (e.g., *v, r, s*) that satisfy both the signature verification and access control conditions. In such cases, the path is deemed logically reachable, and a signature replay vulnerability is reported.

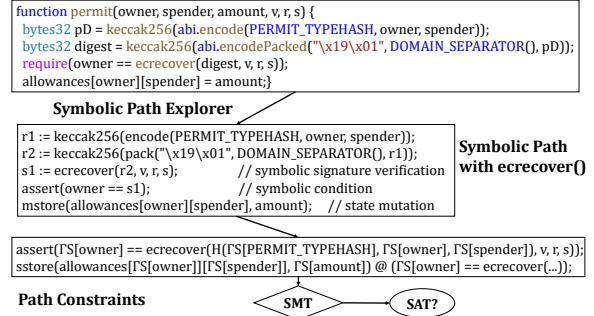


Figure 14: Symbolic Path Constraints for *permit()*.

4.5 Signature Replay Attack Detection

This subsection introduces *domain-specific patterns* derived from various SRVs characteristics to identify hazardous patterns.

Cross-chain Replay Attack (X-CRA). LASiR analyzes the taint propagation path by examining variable reads and writes in *Warnings_nodes* to determine if any sanitization operations exist before reaching the sinks (*ecrecover(_hash, v, r, s)*). It specifically checks whether the *_hash* value is affected by *block.chainid* and tracks the taint propagation of intermediate variables through program dependencies. It determines if the propagation is interrupted by any sanitization operations. If no sanitization is found and the *_hash* is contaminated, LASiR identifies the X-CRA and generates *Warnings*.

Cross-Project Replay Attack (X-PRA). LASiR analyzes variable dependencies in *Warnings_nodes* to confirm if the *_hash* value at the sinks (*ecrecover(_hash, v, r, s)*) is influenced by the *address(this)* variable. Using the same detection logic as X-CRA, LASiR employs program dependencies to check if the *_hash* value depends on *address(this)* and verifies whether intermediate variables (influenced by *address(this)*) have sanitization operations. If the taint propagation path is uninterrupted and there are no sanitization operations, LASiR identifies the presence of X-PRA and generates *Warnings*.

Contract Account Signature Replay (CASR). LASiR first retrieves the code blocks implementing the *isValidSignature()* function [26] from *Warnings_nodes*, as this function is crucial for contract signature verification. It then analyzes contamination at the *ecrecover(_hash, v, r, s)* sinks in these blocks, checking if the *_hash* value depends on *msg.sender* information and whether intermediate variables (influenced by *msg.sender*) are sanitized. If the taint propagation path is complete and lacks sanitization, it identifies the CASR risk and generates corresponding *Warnings*.

Signature State Management Issue (SSMI). To determine if the signature verification process includes verification of the signature

usage state, *LASiR* examines read and write operations affecting the `_hash` value before and after `ecrecover(_hash, v, r, s)`. Specifically, it checks for read-after-write operations for key variables identified in *Step 4*. If these operations are not found, it concludes that there is no check for the signature usage state, identifying SSMI.

Signature Malleability Attack (SMA). *LASiR* analyzes the code blocks preceding `ecrecover(_hash, v, r, s)` to check for statements restricting the `v` and `s` variables. Ensures that $0 < s < secp256k1n \div 2 + 1$ and $v = 27$ or 28 . If these conditions are not met, SMA exists.

5 Evaluation

In this section, we analyze and evaluate the effectiveness of *LASiR* in detecting SRVs by answering the following research questions:

- RQ1: How does *LASiR* perform on the large-scale dataset?
- RQ2: What is the performance of *LASiR* in detecting SRVs?
- RQ3: How does LLM enhance *LASiR*'s effectiveness?

5.1 Experiment Setup

The experiment was carried out on a Ubuntu 20.04.1 LTS server equipped with a 16-core Intel(R) Xeon(R) Gold 5217 processor.

Implementation: *LASiR* is implemented in Python, utilizing *SlithIR* [16] for constructing the I-PDG and dependency analysis, and *Rattle* [15] for Control Flow Graph (CFG) path recovery and symbol execution. Additionally, *LASiR* is developed and tested using the *DeepSeek-V3* [19] LLM API services, known for its robust capabilities in code semantic tasks, offering a 128K input limit and low token price [18]. It employs default parameters, sets *Temperature* to 0 to reduce randomness, and uses a three-request mechanism to ensure result stability by handling abnormal returns.

Datasets: To evaluate *LASiR*'s performance, we selected *Ethereum* [24], *BSC* [58], *Polygon* [59], and *Arbitrum* [3] based on their *Total Value Locked* (TVL) and active user rankings from *DefiLlama* [21], and collected 918,964 contract source codes (2017.10–2024.01). We then constructed two datasets to assess *LASiR*'s effectiveness in detecting SRVs. **DB1:** To evaluate *LASiR*'s performance on large-scale, real-world datasets, we searched for contracts containing `ecrecover()` in their ASTs to identify those involving signature verification. This process yielded 15,383 contracts, distributed across *Ethereum* (4,513), *BSC* (5,590), *Polygon* (4,140), and *Arbitrum* (1,140). **DB2:** To analyze *LASiR*'s accuracy, we manually analyzed 500 randomly selected contract source codes from *DB1*, identifying 72 positive and 428 negative cases.

5.2 RQ1: Performance on large-scale dataset

To evaluate *LASiR*'s performance on large-scale datasets, we conducted experiments on *DB1* dataset. During the experiment, we recorded statistics on the number, proportion, average detection time, and cost across four blockchains.

Table 1: Statistics of Large-Scale Detection Results.

	X-CRA	X-PRA	CASR	SSMI	SMA	Proportion	Avg. Time (s)	Total Cost (\$)
Ethereum	455	493	14	734	674	19.63%	38.27	4.40
BSC	211	198	3	366	252	9.29%	40.67	4.23
Polygon	324	301	3	375	353	7.11%	41.37	5.17
Arbitrum	51	52	2	73	68	5.94%	40.94	1.20

The detailed results in Table 1 show that the higher quantities of SSMI and SMA pose significant threats to signature security.

While CASR is less common, largely due to the adoption of the *EIP-1271* standard [26], which enhances the security of contract account signatures. Notably, 19.63% of contracts that use signatures on *Ethereum* contain SRVs, highlighting a widespread issue. *LASiR*'s average detection time is approximately 40 seconds, with an overall LLM API cost of \$15, showcasing its efficiency and cost-effectiveness for large-scale dataset detection.

To assess the real-world impact of signature reuse vulnerabilities (SRVs), we conducted an analysis of contract addresses with non-trivial asset holdings and manually evaluated their susceptibility to signature reuse. To ensure both meaningful coverage and practical feasibility, we filtered for contracts with non-zero balances, yielding a dataset of 258 contract addresses. Subsequent manual inspection revealed that 31 of these contracts exhibited signature reuse behaviors that could be exploited on forked blockchains, collectively securing approximately \$4.76 million in active assets. Among these, 24 contracts contained signature-verifiable transactions that were replayable by modifying input parameters. We validated their replayability using Tenderly's online simulator, which allowed us to emulate transaction behavior. The remaining seven contracts required the manual construction of proof-of-concept (PoC) exploits to confirm their exploitability. All corresponding simulation links and PoC scripts are publicly available in our code repository. For ethical considerations, our validation strictly focused on confirming the presence of signature reuse behaviors without performing any asset-draining operations.

Answer to RQ1: *LASiR* demonstrates rapid detection and cost efficiency in large-scale analyses. Experimental results show that 19.63% of contracts using signatures contain SRVs on *Ethereum*, with a wide distribution. Manual inspection further reveals that these vulnerabilities affect active assets worth \$4.76 million.

5.3 RQ2: Detection Performance Analysis

To evaluate the performance of *LASiR* in detecting SRVs, we conducted experiments with *DB2* (including 72 positive and 428 negative labels). Furthermore, to analyze the effectiveness of *LASiR*'s LLM-assisted static taint analysis, we conducted comparative experiments with existing tools, covering static analysis, LLM analysis, and the combined approach of LLM and static analysis.

As shown in Table 2, *LASiR* achieved a *Precision* of 82.14%, *Recall* of 95.83%, and an F1-score of 88.46%, effectively detecting SRVs. Meanwhile, we conduct a further analysis of the reasons behind the 15 false positives (FP) and 3 false negatives (FN).

False positives: The primary causes originate from two aspects: Unrelated States Restrictions and Implementation Errors. Our analysis reveals that 11 FPs are due to state restrictions that are unrelated to signature verification. As shown in Figure 15, while the `get()` function includes signature checks, resulting in *SMA*, the balance reset logic (`wallets[signer]! = 0`) at line 4 prevents token transfers, even if signature replay occurs. This also relates to *LASiR*'s strategy of exclusively extracting signature-related states through static taint analysis to filter excessive divergence in LLM output, thereby overlooking unrelated signature state constraints that indirectly limit replay occurrence. Furthermore, the remaining 4 FPs involve developer-customized errors, including flawed verification execution and improper security library usage. Despite meeting SRVs

definitions, signatures cannot be verified due to implementation errors, and such developer-induced errors are frequent.

```

1 function get(bytes32 _r, bytes32 _s, uint8 _v) {
2     require(_v == 27 || _v == 28);
3     address signer = ecrecover(..., _v, _r, _s);
4     require(... && wallets[signer] != 0);
5     payable(msg.sender).transfer(wallets[signer]);
6     wallets[signer] = 0;
7 }
```

Figure 15: Unrelated States Restrictions.

False Negatives: Semantic analysis of the assembly code is limited, affecting the accuracy of the taint propagation. In Figure 16, the `transfer_from()` function parses signature data using the assembly code and verifies it through `ecrecover()`. *LASiR* identifies `ecrecover()` as a sink in static taint analysis, tracking taints from the source (`_data`) to the sink. However, due to the LLM's lack of in-depth learning of smart contract assembly code, it often misinterprets the semantic functions of assembly code, resulting in incorrect taint tracking and subsequently causing false negatives.

```

1 function transfer_from(bytes memory _data) public {
2     assembly { sig_r := mload(_data);
3             sig_s := mload(add(_data, 32));
4             sig_v := ... }
5     ecrecover(limit_hash, sig_v, sig_r, sig_s );
```

Figure 16: Custom Assembly Implementation of `ecrecover()`.

Comparison with existing tools. To further evaluate the effectiveness of the LLM-assisted static taint analysis of *LASiR*, we extended existing general-purpose detection tools (covering static analysis, LLM analysis, and the combination of LLM analysis and static verification) with the definition of SRVs for comparison.

Table 2: Comparison with Existing Tools

	GPT-4o	DeepSeek-R1	DeepSeek-V3	Slither4SRV	Siguard	GPTScan	LASiR
TP	36	50	1	61	3	23	69
FP	222	245	7	375	0	162	15
FN	36	22	73	11	69	49	3
TN	206	183	419	53	418	266	413
Precision	13.95%	16.95%	12.50%	13.99%	100.00%	12.43%	82.14%
Recall	50.00%	69.44%	1.35%	84.72%	4.17%	31.94%	95.83%
F1-score	21.82%	27.25%	2.44%	24.02%	8.00%	17.90%	88.46%
Total Cost (\$)	17.01	4.25	2.82	0	0	0.16	0.3

To select general-purpose tools, we reviewed the top journals, conference papers, and audit reports to search general analysis tools (e.g., [16, 25, 54]). For static analysis tools, we selected *Slither* for its scalable architecture and modular detection framework, thereby extending the development of *Slither4SRV* with SRVs detection rules. Additionally, we incorporated Zhang et al.'s tool, *Siguard*, which supports the detection of SSMI vulnerabilities and represents the first work targeting signature-related vulnerabilities [64]. For LLM analysis, the SRVs definitions were structured as detection prompts and directly analyzed using three LLMs: *GPT-4o* [40], *DeepSeek-R1* [20], and *DeepSeek-V3* [19]. *GPT-4o* reflected benchmark levels, *DeepSeek-R1* excelled in reasoning for complex tasks, and *DeepSeek-V3* served as the base model for *LASiR*, facilitating a comparison of *LASiR* with pure LLM analysis. Additionally, *GPTScan* [51] was chosen for its combined approach of LLM analysis and static verification, with reconstructed properties to support SRVs analysis.

As shown in Table 2, *LASiR* demonstrated outstanding performance, particularly in *Precision* (82.14%) and *Recall* (95.83%). In

contrast, other tools revealed insufficient and unbalanced capabilities. For instance, *Siguard* attained an *F1-score* of only 8%, detecting three true positives vulnerabilities but missing most SRVs, resulting in a *Recall* of just 4.17%. While *Slither4SRV* achieved a high *Recall* of 84.72%, its *Precision* was only 13.99%, reflecting the limitations of pattern-based static analysis. Among the three LLM models, *DeepSeek-R1* performed best with a *Recall* of 69.44% owing to its semantic understanding and reasoning abilities, but it also had the highest number of false positives (245), leading to a *Precision* of 16.95%. This poor performance was mainly due to LLM's limited program analysis capabilities and unstable output. Additionally, *GPTScan4SRV* reduced false positives by verifying the LLM output by static analysis verification. However, the separation of LLM analysis from static analysis introduces challenges. Specifically, the reliance on fixed patterns in static analysis, coupled with inconsistencies in the LLM's interpretation of scenarios and properties, frequently leads to semantic information loss, which adversely affects detection performance. Comparison with existing tools shows *LASiR* efficiently leverages the semantic understanding of the LLM to guide static taint analysis, using code syntax rules to filter LLM outputs in real time. For instance, while *DeepSeek-V3* had a *Recall* of 1.35% and an *F1-score* of 2.44%, *LASiR*'s *F1-score* reached 88.46% when using *DeepSeek-V3* for static taint analysis, a 36-fold improvement. This approach maximizes the strengths of different technologies, providing a robust solution for SRVs detection.

Aanswer to RQ2: *LASiR* detects SRVs leveraging the semantic understanding of LLM to aid in static taint analysis. Compared to existing methods, its *F1-score* for SRVs detection reached 88.46%, demonstrating excellent performance.

5.4 RQ3: Impact of LLM on Performance

To analyze the impact of LLM on enhancing *LASiR*'s detection performance, we conducted ablation experiments with four groups based on the participation of LLM in different phases of detection. The groups were divided as follows: *LLM_Phases1&2&3* (participated in all three phases), *LLM_Phases1&2* (participated in Phases 1 and 2 only), *LLM_Phase1* (participated in Phase 1 only), and *No_LLM* (did not participate in any phase). The experimental data, *DB2*, included 500 contracts with 72 positive and 428 negative labels.

Table 3: Statistics of LLM's Impact on Different Phases

	TP	FP	TN	FN	Precision	Recall	F1-score	Time (s)
LLM_Phases1&2&3	69	15	413	3	82.14%	95.83%	88.46%	41.03
LLM_Phases1&2	35	45	383	37	43.75% ↓	48.61% ↓	46.05% ↓	71.35 ↑
LLM_Phase1	31	351	77	41	8.12% ↓	43.06%	13.66%	55.27
No_LLM	19	413	15	53	4.40% ↓	26.39% ↓	7.54% ↓	15.49 ↓

Table 3 shows the statistical results of LLM participation in different phases of *LASiR*, highlighting significant effects in each phase. Comparing *LLM_Phases1&2&3* with *LLM_Phases1&2*, we observe that without LLM guidance in Phase 3, detection metrics decrease by about half. This is because, during symbolic execution, the lack of LLM guidance greatly expands the path search space, causing many irrelevant functions to be traversed, and leading to many missed vulnerabilities and a significant increase in FNs. Comparing *LLM_Phases1&2* with *LLM_Phase1*, the absence of LLM in Phase 2's sanitization checks results in an incomplete state at the

sinks, increasing FPs and dropping *Precision* from 43.75% to 8.12%. These results underscore LLM's significant role in different phases.

Furthermore, *LASiR* effectively leverages the semantic understanding of LLM to improve detection accuracy and completeness. Comparing *No_LLM* with *LLM_Phases1&2&3*, *Precision* improved from 4.40% to 82.14%, *Recall* from 26.39% to 95.83%, and *F1-score* from 7.54% to 88.46%. These results showcase LLM's effectiveness in improving vulnerability detection accuracy, reducing false positive rates, and narrowing the symbolic execution search space.

To evaluate the LLM's impact on detection efficiency, we compared detection times in four groups. The results of Table 3 showed that with LLM throughout the process (*LLM_Phases1&2&3*), detection time was 41.03 seconds, demonstrating balanced performance. Without LLM guidance from *Phase 3* (*LLM_Phases1&2*), detection time increased to 71.35 seconds due to symbolic execution path explosion. Without LLM support (*No_LLM*), most dangerous paths were ignored, reducing the detection time to 15.49 seconds but significantly lowering the accuracy and reliability.

Answer to RQ3: The semantic information from LLM, through contract context analysis, is crucial for *LASiR*'s detection process. This enhances static taint analysis by contract semantic understanding, improving accuracy and efficiency.

6 Discussion

6.1 Case Study

We classify this case as *Whitelist Signature Permission Abuse* because these NFT contracts incorrectly verify whitelist signatures, allowing a single signature to mint multiple NFTs, with assets totaling \$24,838. The *Sol Flowers x AP* NFT contract [61] allows whitelisted users to mint multiple NFTs using the same signature. The process is as follows: ① The attacker completes an offline task. ② After completing the task, the attacker requests a signature to mint an NFT. ③ The attacker obtains the signature. ④ The attacker submits a minting request to the contract and mints an NFT. ⑤ The attacker reuses the signature to submit another minting request and mints another NFT. Due to the contract's lack of signature usage management (i.e., it does not check if the signature has been verified), there is SSMI. This allows the attacker to reuse signatures to mint multiple NFTs, including high-rarity ones. This signature replay vulnerability severely impacts the assets of the *Sol Flowers x AP*.

6.2 Threats to Validity

Internal Validity: One internal threat lies in the information asymmetry between static analysis and LLM. To ensure a structured output and further analysis, key variables from the LLM output may lack the context required for static analysis. To mitigate this, *LASiR* analyzes dependencies of key variables within the I-PDG, gathering the relevant code blocks to provide necessary context. Furthermore, in *Phase 3*, *LASiR* conducts self-verification using symbolic execution to confirm path reachability, ensuring reliability. Another internal threat is the extension of *Slither* and *GPTScan* to support SRVs detection. After reviewing the state-of-the-art tools, we selected them because they are general-purpose tools with excellent extensibility. According to SRVs definitions, *GPTScan* can directly migrate by modifying natural language properties and scenario descriptions. Additionally, *Slither* can be extended using *SlithIR* intermediate languages within its modular detection framework.

External Validity: One external threat is the generality of vulnerability definitions. To ensure that SRVs reflect real-world issues, we collected and analyzed a total of 1,419 open-source security audit reports from 37 security teams. This is the first work to define and detect *Signature Replay Vulnerabilities* in smart contracts. These types of vulnerabilities represent real issues encountered during development, and all identified vulnerabilities were sourced from authentic security reports.

7 Related Work

7.1 Defining and Detecting Bugs in Contracts

In recent years, the discovery and detection of vulnerabilities in smart contracts have become a key focus in blockchain security. As developers' awareness and mitigation technologies advance, common vulnerabilities like *Integer Overflow* [47] and *Unchecked Return Values* [49] have significantly decreased in real production environments. However, continuous innovation in DeFi applications has led to new vulnerabilities characterized by more complex states and divergent execution paths. To define and detect these vulnerabilities, existing work has summarized specific vulnerability types by investigating data from real production environments. For example, Chen et al. [7] investigated *Ethereum StackExchange* posts and actual contracts, summarizing 20 common code defects and highlighting five high-risk protocol errors. Zhang et al. [63] studied *Ethereum* transactions, contracts, and *StackExchange* posts, identifying five major obstacles in developers' cryptographic tasks and providing a practical guide to improve the development experience. Yang et al. [62] analyzed *StackOverflow* posts, defined five common defects in NFT contracts, and proposed the *NFTGuard* symbolic execution tool *NFTGuard* to automatically detect these defects.

7.2 LLMs for Smart Contract Security Research

With the development of LLM technology, its importance in smart contract security research has grown significantly. *GPTScan* by Sun et al. [51] uses the understanding of *Generative Pre-training Transformer* (GPT) [40] to identify vulnerabilities by decomposing logical vulnerability types into scenarios and attributes, guiding GPT to match vulnerabilities. Wang et al. [55]'s *SmartInv* automatically detects "machine unauditables bugs" by reasoning about multimodal information, like source code and natural language, and uses the *Tier of Thought* (ToT) prompting strategy to generate and analyze invariants to detect. Liu et al. [35] utilized state-of-the-art LLMs like *GPT-4* to derive customized properties for unknown code from existing manual properties, such as audit reports. They created *PropertyGPT* to verify the correctness of these properties.

8 Conclusion

This paper presents the first empirical study on SRVs and defines five types. We designed *LASiR*, which leverages LLMs to assist static taint analysis and integrates symbolic execution verification for efficient detection of SRVs. To evaluate detection performance, we collected 918,964 smart contracts on multiple blockchains. The results show that \$4.76 million in active assets are affected, with 19.63% of *Ethereum* contracts containing SRVs. Manual verification indicates *LASiR* achieves a 95.83% *Recall*, with *Precision* and *F1-score* of 82.14% and 88.46%, respectively. LLMs significantly improve *LASiR*'s performance, enhancing its detection capabilities.

Acknowledgments

This work was supported in part by the National Key Research and Development Program of China (No. 2023YFB2703600), the National Natural Science Foundation of China (Nos. 62276279, 62306344), the Guangdong Basic and Applied Basic Research Foundation (Nos. 2024B1515020032, 2024A1515010253), and the Major Key Project of Peng Cheng Laboratory under Grant PCL2025A07.

References

- [1] Solidity Academy. 2023. Demystifying Signature Malleability Attacks: A Deep Dive into Blockchain Security. <https://medium.com/@solidity101/demystifying-signature-malleability-attacks-a-deep-dive-into-blockchain-security-6c7c8e6d25ac>
- [2] ahmetw.eth. 2024. Anatomy Solidity: Multi Signature Wallets Explained. <https://medium.com/coinmonks/anatomy-solidity-how-to-work-multi-signature-wallet-a4fa49e70dec>.
- [3] Arbitrum. 2025. Arbitrum — The Future of Ethereum. <https://arbitrum.io/>.
- [4] 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
- [5] Remco Bloemen, Leonid Logvinov, and Jacob Evans. 2017. EIP-712: Typed Structured Data Hashing and Signing. <https://eips.ethereum.org/EIPS/eip-712>. Ethereum Improvement Proposals, no. 712, September 2017.
- [6] Vitalik Buterin, Yoav Weiss, Dror Tiros, Shahaf Nacson, Alex Forsthat, Kristof Gazso, and Tjaden Hess. 2021. ERC-4337: Account Abstraction Using Alt Mempool. Ethereum Improvement Proposal. <https://eips.ethereum.org/EIPS/eip-4337>
- [7] 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
- [8] code423n4. 2021. Signature replay attacks for different identities (nonce on wrong party). <https://github.com/code-423n4/2021-10-ambire-findings/issues/39>.
- [9] code423n4. 2022. Cross-Chain Signature Replay Attack. <https://github.com/code-423n4/2023-01-biconomy-findings/issues/466>.
- [10] Stack Exchange Community. 2021. What is an EOA account? Ethereum Stack Exchange. <https://ethereum.stackexchange.com/questions/5828/what-is-an-eoa-account>
- [11] Consensys. 2021. 4.21 TransactionManager - Adherence to EIP-712. <https://diligence.consensys.io/audits/2021/07/connext-nxtp-noncustodial-xchain-transfer-protocol/#transactionmanager--adherence-to-eip-712>.
- [12] Coders Errand contributors. 2024. ECREcover Signature Verification in Ethereum. <https://coders-errand.com/ecrecover-signature-verification-ethereum/>.
- [13] Wikipedia contributors. 2024. Digital signature – Wikipedia, The free encyclopedia. https://en.wikipedia.org/wiki/Digital_signature.
- [14] corwintines. 2024. ERC-20 Token Standard. <https://ethereum.org/en/developers/docs/standards/tokens/erc-20/>.
- [15] cryptic. 2025. Rattle. <https://github.com/cryptic/rattle>.
- [16] cryptic. 2025. SlithIR. <https://github.com/cryptic/slither/wiki/SlithIR>.
- [17] Cyfrin. 2025. Solodit Bug Bounty Platform. <https://solodit.cyfrin.io/bug-bounties>.
- [18] DeepSeek. 2024. Introducing DeepSeek-V3. <https://api-docs.deepseek.com/news/news1226>.
- [19] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] <https://arxiv.org/abs/2412.19437>
- [20] DeepSeek-AI. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs.CL] <https://arxiv.org/abs/2501.12948>
- [21] DefiLlama. 2025. DefiLlama. <https://defillama.com/docs/api>.
- [22] eclairCKBot. 2024. Scep256k1: A Key Algorithm in Cryptocurrencies. [https://www.nervos.org/knowledge-base/secp256k1_a_key%20algorithm_\(explainCKBot\)](https://www.nervos.org/knowledge-base/secp256k1_a_key%20algorithm_(explainCKBot)).
- [23] Etherscan. 2024. Smart Contracts Audit and Security. https://etherscan.io/directory/Smart_Contracts/Smart_Contracts_Audit_And_Security.
- [24] Etherscan. 2025. Ethereum (ETH) Blockchain Explorer. <https://etherscan.io/>.
- [25] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2022. eTainter: detecting gas-related vulnerabilities in smart contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (Virtual, South Korea) (ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 728–739. doi:10.1145/3533767.3534378
- [26] Francisco Giordano, Matt Condon, Philippe Castonguay, Amir Bandeali, Jorge Izquierdo, and Bertrand Masius. 2018. ERC-1271: Standard Signature Validation Method for Contracts. <https://eips.ethereum.org/EIPS/eip-1271>. Ethereum Improvement Proposals, no. 1271, July 2018.
- [27] Yoni Goldberg. 2023. EIP-1271: Signature Verification for Smart Contract Wallets. <https://www.dynamic.xyz/blog/eip-1271>.
- [28] Hermez. 2020. _checkSig allows signature re-use. <https://solodit.cyfrin.io/issues/checksig-allows-signature-re-use-trailofbits-hermez-pdf>.
- [29] Howy Ho. 2024. ERC-1271 Signature Replay Vulnerability. <https://www.alchemy.com/blog/erc-1271-signature-replay-vulnerability>.
- [30] Immunefi. 2021. Polygon Double-Spend Bugfix Review — \$2m Bounty. <https://medium.com/immunefi/polygon-double-spend-bug-fix-postmortem-2m-bounty-5a1db09db7f1>.
- [31] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future. arXiv:2408.02479 [cs.SE] <https://arxiv.org/abs/2408.02479>
- [32] Peter Jihoon Kim, Kevin Britz, and David Knott. 2020. ERC-3009: Transfer With Authorization [DRAFT]. <https://eips.ethereum.org/EIPS/eip-3009>. Ethereum Improvement Proposals, no. 3009, September 2020.
- [33] LASiR. 2025. Online supplement material. <https://anonymous.4open.science/r/LASiR-B207>.
- [34] Zewei Lin, Jiachi Chen, Jiajing Wu, Weizhe Zhang, Yongjuan Wang, and Zibin Zheng. 2024. CRPWarner: Warning the Risk of Contract-Related Rug Pull in DeFi Smart Contracts. *IEEE Transactions on Software Engineering* 50, 6 (2024), 1534–1547. doi:10.1109/TSE.2024.3392451
- [35] Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. 2025. PropertyGPT: LLM-driven Formal Verification of Smart Contracts through Retrieval-Augmented Property Generation. (2025). doi:10.14722/ndss.2025.241357
- [36] Fuchen Ma, Zhenyang Xu, Meng Ren, Zijng 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
- [37] Metalex. 2025. Incorrect Function Signature in DAOvoteGrantImplplantProposeAdvancedGrant. <https://solodit.cyfrin.io/issues/incorrect-function-signature-in-daovotegrantimplplantproposeadvancedgrant-daovetograntimplplantproposeadvancedgrant-mixbytes-none-metalex-markdown>.
- [38] Neptune Mutual. 2024. Understanding ERC-20 Permit and Associated Risks. <https://neptunemutual.com/blog/understanding-erc-20-permit-and-associated-risks/>.
- [39] Trail of Bits. 2025. Publications. <https://github.com/trailofbits/publications>.
- [40] OpenAI. 2022. Introducing ChatGPT. <https://openai.com/index/chatgpt/>.
- [41] OpenZeppelin. 2025. OpenZeppelin Contracts. <https://github.com/OpenZeppelin/openzeppelin-contracts>
- [42] Sigma Prime. 2022. Direct usage of ecrecover() allows signature malleability. <https://solodit.cyfrin.io/issues/direct-usage-of-ecrecover-allows-signature-malleability-sigmaprime-none-interest-protocol-pdf>.
- [43] Fintech Review. 2024. Cross-Chain Interoperability: The Future of Blockchain Networks. <https://fintechreview.net/cross-chain-interoperability-the-future-of-blockchain-networks/>.
- [44] RugDocWiki. 2024. Introduction to Ethereum's Keccak-256 Algorithm. <https://wiki.rugdoc.io/docs/introduction-to-ethereums-keccak-256-algorithm/>.
- [45] Philipp Schmid. 2023. How to Prompt Llama 2. <https://huggingface.co/blog/llama2#how-to-prompt-llama-2>.
- [46] SharkTeam. 2023. Analysis of the AzukiDAO Attack Incident. <https://x.com/sharkteamorg/status/1676892088930271232>.
- [47] Shashank. 2022. Integer Overflow and Underflow in Smart Contracts. <https://blog.solidityscan.com/integer-overflow-and-underflow-in-smart-contracts-9598032b5a99>.
- [48] SlowMist. 2025. Knowledge-Base. <https://github.com/slowmist/Knowledge-Base>.
- [49] Sm4rty. 2022. Unchecked Call Return Value|Solidity Security #1. <https://sm4rty.medium.com/unchecked-call-return-value-solidity-security-1-fe794a7cd6f>.
- [50] Donn Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.
- [51] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (Lisbon, Portugal) (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 166, 13 pages. doi:10.1145/3597503.3639117
- [52] BlockSec Team. 2025. Audit Reports. <https://github.com/blocksecteam/audit-reports>.
- [53] QuillAudits Team. 2024. Decoding Azuki DAO Hack. <https://www.quillaudits.com/blog/hack-analysis/azuki-dao-hack>.
- [54] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (Toronto, Canada) (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 67–82. doi:10.1145/3243734.3243780
- [55] 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
- [56] 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
- [57] Wikipedia. 2025. Elliptic Curve Digital Signature Algorithm. https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm.
 - [58] Wikipedia contributors. 2025. Binance — Wikipedia, The Free Encyclopedia. <https://en.wikipedia.org/w/index.php?title=Binance&oldid=1214652916>. [Online; accessed 10-March-2025].
 - [59] Wikipedia contributors. 2025. Polygon (blockchain) — Wikipedia, The Free Encyclopedia. [https://en.wikipedia.org/w/index.php?title=Polygon_\(blockchain\)&oldid=1214411541](https://en.wikipedia.org/w/index.php?title=Polygon_(blockchain)&oldid=1214411541). [Online; accessed 10-March-2025].
 - [60] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.
 - [61] Sol Flowers x AP. 2025. Sol Flowers x AP NFT contract. <https://etherscan.io/address/0x3f491600E8B81805CA8e11361155A8c49B0E2be4#code>.
 - [62] 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
 - [63] Jiahuo 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
 - [64] Jiahuo Zhang, Yue Li, Jianbo Gao, Zhi Guan, and Zhong Chen. 2023. Siguard: Detecting Signature-Related Vulnerabilities in Smart Contracts. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 31–35. doi:10.1109/ICSE-Companion58688.2023.00019
 - [65] Jiahuo Zhang, Yiming Shen, Jiachi Chen, Jianzhong Su, Yanlin Wang, Ting Chen, Jianbo Gao, and Zhong Chen. 2025. Demystifying and Detecting Cryptographic Defects in Ethereum Smart Contracts . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 114–126. doi:10.1109/ICSE55347.2025.00010