From Patterns to Precision: LLM-Guided Detection of Signature Verification Flaws in Smart Contracts

Huixin Wang*†, Kailun Yan‡(✉), and Wenrui Diao*†(✉)
*School of Cyber Science and Technology, Shandong University
wanghuixin@mail.sdu.edu.cn, diaowenrui@link.cuhk.edu.hk
†State Key Laboratory of Cryptography and Digital Economy Security, Shandong University
‡Institute for Advanced Study, Tsinghua University, kailun@mail.tsinghua.edu.cn

***Abstract*—Off-chain Signing and On-chain Verification (OSOV) is a widely adopted contract pattern that enhances user experience. However, existing studies primarily focus on isolated defects, lacking a systematic understanding of the fundamental security requirements of OSOV. By analyzing OSOV workflows and real-world implementations, this paper identifies four security-critical fields in signed messages that together define the essential security boundary. In practice, these fields exhibit diverse implementation styles, making it difficult for rule-based inspection to achieve comprehensive and accurate detection. To address this challenge, we propose a two-stage LLM-based summarization approach that automatically extracts implementation patterns of these fields from large-scale signature verification functions (VFs) and uses the summarized patterns to guide the construction of detection rules. We implement these rules in a fine-grained static analysis tool, VCSCOPE, and evaluate it on 22,374 real-world VFs. Experimental results demonstrate that mainstream LLMs effectively perform the field summarization task and that our approach reduces token consumption by 74.3%. VCSCOPE achieves 98.7% precision, 93.1% recall, and a 95.8% F1-score, revealing widespread security risks in current OSOV.**

I. INTRODUCTION

Off-chain Signing and On-chain Verification (OSOV) is a common pattern in decentralized applications (Dapps), allowing users to authorize operations by signing messages off-chain and submitting them on-chain for verification and execution. OSOV has powered a wide range of innovations in decentralized systems, including tokens [9], [10], [12], NFT marketplaces [19], and decentralized voting and governance [15]. In decentralized exchanges (DEXs), OSOV enables off-chain signed orders to be executed only when matched on-chain. As of December 2024, DEXs reached a record monthly trading volume of $462 billion [2], with Uniswap [18] accounting for over $100 billion. Similarly, NFT platforms like OpenSea [1], peaking at $3.4 billion monthly, rely on OSOV to facilitate off-chain listings and seamless on-chain execution without requiring sellers to stay online. Beyond trading, OSOV also supports off-chain voting [17] and governance [4], where signed messages are batch-submitted on-chain to improve scalability and reduce costs.

Recently, researchers have begun to recognize the importance of OSOV, leading to the development of various analysis tools [36], [54], [45], [50], [31]. However, existing work has not systematically examined the OSOV workflow. Most studies lack a clear threat model and focus only on individual flaws within signature verification. Furthermore, these detection tools are typically built upon manual insights or heuristic rules, making it difficult to effectively capture the range of diverse implementations deployed in practice.

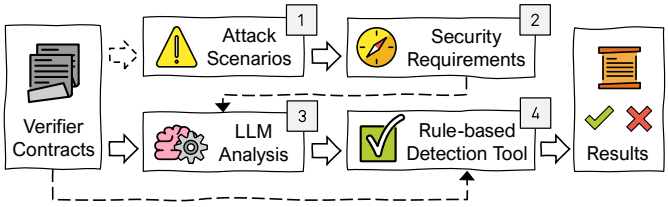

Fig. 1: Overview of Our Detection Framework

Our Work. This work conducts a comprehensive analysis of signature verification security in smart contracts. We define a threat model and three attack scenarios covering both on-chain and off-chain workflows. Based on this model, we identify four security-critical fields that together form the minimal security boundary of any verification function (VF). We utilize a Large Language Model (LLM) as an automated auditor to mine and summarize implementation patterns of these fields, which in turn guide the construction of a rule-based detection tool. Finally, we develop a static analysis tool to automatically detect and validate the implementation correctness of security-critical fields in VFs. Figure 1 presents an overview of our detection framework.

Specifically, we employ LLMs to mine implementation patterns of security-critical fields in VFs. We first extract structural features from the signature-verification segment of each VF as its fingerprint, and group functions sharing the same fingerprint. For each group, only one representative sample is analyzed by the LLM, avoiding redundant analysis due to code reuse. The LLM then extracts implementation patterns of four security fields, followed by a two-stage summarization to merge similar patterns. Experiments show that five state-of-the-art LLMs capture over 81% of patterns, and the summarized representative patterns achieve 100% coverage of all mined patterns. This demonstrates the strong capability of LLMs in mining and summarizing code patterns. Moreover, our grouping strategy reduces token consumption by 74.3%.

Based on the patterns extracted by LLMs, we design seven detection rules and develop VCSCOPE (Verifier Contract

Scope), a fine-grained static analysis tool. The core idea of our detection approach is to classify variables involved in the signature verification process into trusted and untrusted sets. The tool then checks whether the untrusted variables satisfy the constraints defined by the detection rules, thereby verifying whether the security-critical fields are correctly implemented. We apply VCSCOPE to 22,374 real-world verification functions (VFs). While 80.5% of on-chain VFs verify the signer, only 41.0% include a nonce and 33.4% incorporate the chainId, leaving them partially exposed to replay attacks. Moreover, 40.9% omit the verifying contract address, enabling potential cross-contract replays.

Finally, we compare our rule-based static tool with the LLM-based detection approach. The rule-based method achieves higher precision (98.7% vs. 88.5%), recall (93.1% vs. 72.8%), and F1-score (95.8% vs. 80.0%). Further analysis shows that our static tool excels at tracking inter-procedural control and data flows, enabling more accurate identification of implicit verification logic. Nevertheless, the semantic understanding capability of LLMs allows them to handle complex cases that are difficult to formalize through rules.

Contributions. The main contributions are as follows:

- *Boundary Definition of Secure Verification.* We investigate OSOV workflows and real-world scenarios to define the minimal security boundary, identifying four security-critical fields that should be enforced in all verification functions.
- *LLM-based Pattern Mining.* We propose an LLM-based approach to mine implementation patterns of four security-critical fields, using fingerprint-based grouping and two-stage summarization for efficient and comprehensive analysis. Experiments confirm that state-of-the-art LLMs effectively accomplish this task.
- *Static Detection Tool.* We develop a fine-grained static analysis tool with seven LLM-guided rules to verify the correct implementation of security-critical fields, achieving high precision in practice.
- *Large-scale Measurement.* We conduct a large-scale measurement to assess the implementation of security-critical fields across 22,374 real-world verification functions, revealing widespread security risks.

II. BACKGROUND

This section provides the necessary background related to signature verification within smart contracts.

A. Off-chain Signing with On-chain Verification (OSOV)

OSOV is a widely adopted pattern in smart contract design [23], [32]. In this scenario, users sign intent messages off-chain using their private keys, and contracts verify the signatures on-chain before executing actions. This design eliminates the need for on-chain approval state management and multiple transactions, enabling gasless approvals and deferred execution. OSOV is extensively used across decentralized applications (Dapps), including token exchanges [18], NFT marketplaces [14], and governance platforms. It supports common features such as delegated tokens or NFT transfers, airdrop claims, DAO voting and delegation [29], [49]. In practice, signature verification is encapsulated in a dedicated Verifier Contract (VC), which contains the verification functions (VFs) that check signed messages.

B. Digital Signatures in Ethereum

Digital signatures rely on asymmetric cryptography: users sign messages with private keys, and anyone can verify the signatures using the corresponding public keys to confirm message integrity and authenticity. In Ethereum, signatures use the Elliptic Curve Digital Signature Algorithm (ECDSA) [21]. Solidity provides a built-in function, `ecrecover` [13], which takes a message hash and a signature `(v, r, s)`, and returns the address that created the signature. This recovered address is commonly used to enforce access control in smart contracts. However, ECDSA signatures are malleable, meaning different signatures can be valid for the same message. To prevent this, Ethereum requires the `s` value in the signature to be in the lower half of the elliptic curve's order [6].

To ensure consistent and secure signature usage, the Ethereum community has proposed several standards and libraries. EIP-712 [7] defines typed structured data signing with domain separation to prevent cross-context replay attacks. EIP-4494 [12] extends the NFT standard (EIP-721 [11]) with a `permit` function for gasless approvals. The OpenZeppelin Contracts library [16] also provides reusable implementations of these standards for safer and simpler signature verification.

III. THREAT ANALYSIS AND SECURITY BOUNDARY

This section first presents the OSOV workflow, then introduces the threat model and three attack scenarios, and finally defines a minimal security boundary.

A. Motivating Example

The OSOV pattern allows users to authorize sensitive actions off-chain. Figure 2 shows a typical NFT *transfer* using EIP-4494 [12], where Alice authorizes Bob to transfer her NFT via an off-chain signature, and Bob submits it on-chain. The workflow involves three steps:

Step 1. Off-Chain Signing. Alice signs a message (*Msg*) authorizing the transfer and sends it with the signature (*Sig*) to Bob through an off-chain channel.

Step 2. Transaction Submission. Bob can optionally verify the signature off-chain using a read-only function from the VC, avoiding gas costs. If valid, he submits an on-chain transaction, paying the gas himself, to call the contract's verification function (VF) with the signed message as input.

Step 3. On-Chain Verification. The contract verifies the message and signature, and if valid, transfers the NFT. The change is recorded on-chain upon transaction confirmation.

The OSOV workflow involves four main entities, each with a distinct role in the authorization and execution process:

- **Signer (Alice):** The asset owner who authorizes actions by signing structured messages with a private key. Alice performs all operations entirely off-chain.

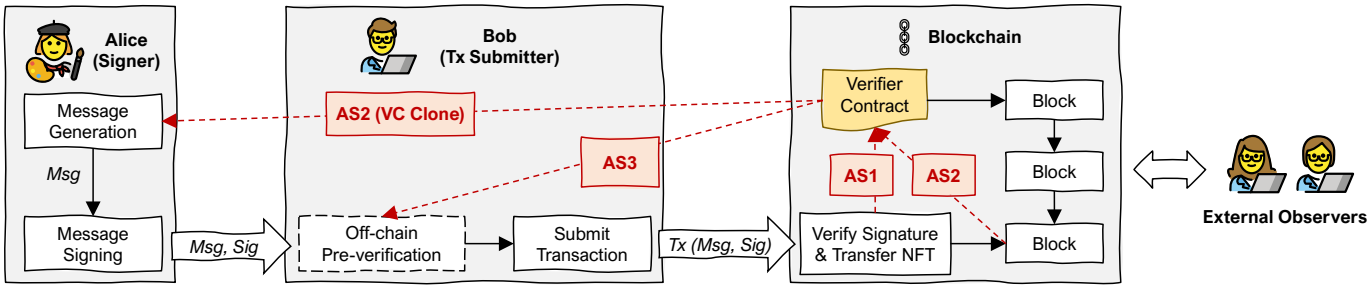

Fig. 2: The Process of Off-Chain Signing and On-Chain Verification (OSOV)

- **Submitter (Bob):** The recipient who submits the signature to the VC for on-chain execution. Bob may be the counterparty or a service platform. He covers the gas cost and may optionally verify the signature off-chain before submission.
- **Verifier Contract (VC):** A smart contract that verifies signatures and executes authorized actions. It contains an on-chain verification function (VF) and may expose a view function (e.g., ERC-1271 [8]) for off-chain verification.
- **External Observers:** Third parties with access to public blockchain data, such as bots, traders, indexers, or attackers. They can monitor signed messages and transactions and may attempt to exploit weaknesses in the OSOV design.

B. Threat Model

We assume the underlying blockchain and standard cryptographic primitives, such as digital signatures, are secure. Misuse stems from flaws in contract logic rather than weaknesses in the cryptography itself. Specifically, adversaries act as regular blockchain users. They can access public on-chain data, send transactions, deploy contracts, and locally simulate verification logic. We consider two types of adversaries: *Submitters* and *External Observers*. These actors may exploit weaknesses in the OSOV workflow or VC logic to bypass verification, perform unauthorized actions, or gain unfair advantages. Such attacks can lead to asset losses for the VC or the signer. In some cases, even the submitter may become a victim when the adversary is an external observer.

C. Attack Scenarios

Prior work has mainly focused on flaws in verification logic, with limited analysis of how these issues lead to real-world attacks in OSOV workflows. Here, we present three common attack scenarios in OSOV workflows: two on-chain (AS1 and AS2) and one off-chain (AS3).

AS1: Signature Forgery. If the VC fails to correctly verify signatures or match signers with the expected addresses, attackers may forge or misuse signatures to bypass checks [13], leading to unauthorized execution of sensitive operations.

AS2: Replay Attacks and VC Cloning. Traditional replay attacks reuse signed messages across different execution contexts, including single-contract, cross-contract, and cross-chain scenarios. We identify a variant in which an attacker deploys a clone of the VC and tricks the signer (e.g., Alice) into signing on the clone. If the original contract does not bind the message to its own address, the attacker can reuse the signature on the original contract to bypass verification.

AS3: Pre-Verification Exploit. Some VCs offer read-only pre-verification, allowing users to check signatures before submitting transactions. If this function shares flaws from AS1 or AS2, attackers can mislead the submitter (e.g., Bob) into accepting invalid or replayed signatures, potentially causing failed transactions or economic loss.

D. Security Boundary

Building on the prior threat model and attack scenarios, we surveyed secure implementations such as OpenZeppelin [16] and identified four critical fields forming the minimal security boundary for OSOV. These fields must be enforced by all VCs to prevent AS1 and AS2, and by any off-chain pre-verification logic to mitigate AS3. They jointly ensure three essential properties: (1) Authenticity: verifies the message origin; (2) Non-reusability: binds the signature to a specific state or context; (3) Unforgeability: prevents valid signatures from being forged without the private key.

```
contract NFTPermit{
  mapping(address => uint) public nonces;
  mapping(uint => address) public owners; // NFT ownership

  function permit(address from, address to, uint tokenId,
       uint nonce, uint deadline, uint8 v, bytes32 r,
       bytes32 s) public {

    require(block.timestamp <= deadline, "Permit expired");
    require(nonces[tokenId] == nonce, "Invalid nonce");

    bytes32 structHash = keccak256(abi.encodePacked("
         PermitNFT", from, to, tokenId, nonce, deadline,
         block.chainid, address(this)));
    bytes32 msgHash = structHash.toEthSignedMessageHash();

    address signer = ecrecover(msgHash, v, r, s);
    require(signer != address(0), "Invalid signature");
    require(signer == from, "Invalid signature");

    nonces[tokenId] += 1; // Invalidate the used nonce

    // Execute token transfer
    require(owners[tokenId] == from, "Not token owner");
    owners[tokenId] = to; // Update ownership
  }
}
```

Listing 1: Example of Verifier Contract.

Signer. Ensures the message originates from the legitimate asset owner. Without verification, attackers can forge signatures (AS1). In Listing 1, `ecrecover` extracts the `Signer` and checks if it matches from (lines 13 to 15), ensuring the request comes from the rightful asset owner.

Nonce. A unique identifier for each message, used to prevent replay attacks by enforcing one-time use. It protects against single-contract replay attacks, as described in AS2. In Listing 1, each NFT (`tokenId`) has a corresponding `Nonce` stored in a mapping. The contract verifies the submitted `Nonce` (line 8) and increments it after execution (line 17), ensuring each message is bound to a specific state and cannot be reused.

VC Address. Binds the signed message to a specific contract, preventing attackers from using cloned contracts to harvest signatures for replay on the original contract (AS2). Without it, cross-contract replays are possible even if nonces match, as attackers can craft messages targeting the original contract.

ChainId. Binds the signed message to a specific blockchain, preventing cross-chain replay attacks, as seen in AS2. Listing 1 includes `block.chainid` in the message hash, ensuring signatures are valid only on the intended chain.

Additional fields (e.g., `deadline`) are excluded from the minimal boundary for two reasons: (1) they serve application-specific purposes rather than core verification security; (2) their integrity is already ensured by the four fields.

IV. LLM-based Field Pattern Mining Framework

This study aims to evaluate whether existing OSOV implementations satisfy the minimal security requirements by examining whether their verification contracts (VCs) correctly implement the four critical security fields. We first analyze the limitations of existing methods and then introduce an LLM-based method for variant mining of security fields. Based on the mined results, we design a static detection tool to evaluate real-world OSOV implementations in Section V.

A. Overview

In practice, the four security fields exhibit substantial implementation diversity. Developers may adopt different variable names, data structures, or control logic to achieve the same semantics. For example, the `nonce` field may appear as a global integer variable (e.g., `globalNonce`) controlling all signatures, or as a mapping (e.g., `nonces`) maintaining a separate counter for each address. Such diversity makes it difficult for existing static detection methods [26], [38] to achieve comprehensive coverage, as their handcrafted rules often rely on pattern matching and prior experience rather than semantic understanding.

To address these challenges, we exploit the semantic reasoning capabilities of large language models (LLMs) to comprehensively summarize real-world implementations of security-critical fields, thereby enabling the formulation of more accurate detection rules. As illustrated in Figure 3, our approach consists of several stages. We first apply Verification Function Grouping (Section IV-B) based on signature logic fingerprints to eliminate unrelated code and duplicates, yielding a concise

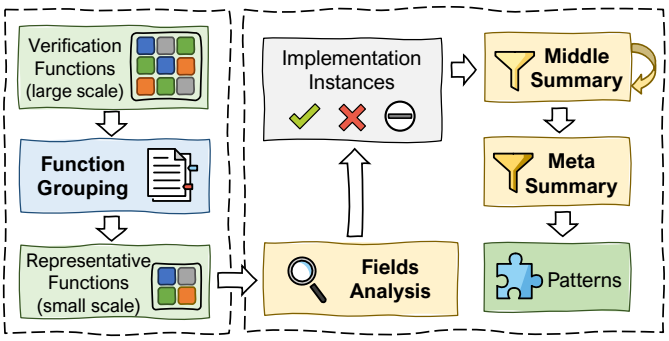

Fig. 3: The Workflow of LLM-Based Variant Mining

set of implementation variants. Next, the LLM acts as an automated contract auditing expert to examine the presence and correctness of each security field and to extract relevant code snippets (Section IV-C). We then employ a two-stage summarization process to generalize the results: (1) the middle summary condenses similar implementations to reduce redundancy; and (2) the meta summary performs higher-level abstraction to unify semantically equivalent implementations and reveal common field-level patterns. Finally, these patterns capture real-world implementation diversity and form a finite, analyzable set that guides the development of our static detection tool in Section V.

B. Verification Function Grouping

Our investigation reveals that verification functions (VFs) exhibit substantial diversity yet frequent repetition. On the one hand, even when the security function is identical, developers often differ in variable naming, encapsulation, and data handling. For instance, the `Nonce` field may be linked to a `tokenId`, bound to the `from` address, or implemented as a global counter. On the other hand, many contracts reuse nearly identical verification logic, often derived from shared libraries or duplicated across multiple functions, differing only in their application-specific business logic. Analyzing each VF in isolation is therefore inefficient. Such redundancy not only increases computational overhead but also diffuses the LLM's attention, potentially leading it to overlook rare but valid implementation variants.

To balance coverage and efficiency, we extract and group the core verification logic using structural fingerprints, producing a compact yet representative set of implementations. This approach effectively reduces redundancy while preserving semantic diversity. As shown in Section VI, our method compresses the dataset to less than 25% of its original size. The grouping process involves four key steps:

- **Step 1. Function Parsing.** This step parses each VF into its Abstract Syntax Tree (AST) and generates its Intermediate Representation (IR). During generation, the analysis recursively expands external calls to obtain their IR, ensuring complete coverage of the verification logic.
- **Step 2. Verification Logic Identification.** This step performs backward analysis on the IR to locate cryptographic

primitives such as `ecrecover`, then traces upward to extract the code slice responsible for signature verification.

- **Step 3. Structure Extraction.** This step simplifies the extracted slice by removing irrelevant instructions and discarding names, retaining only types and data dependencies to produce a canonical logic structure.
- **Step 4. Fingerprinting and Grouping.** The final step computes the `SHA-256` hash of the code structure as the function's fingerprint. All functions with identical fingerprints are grouped together.

C. Field Pattern Analysis via LLMs

We employ LLMs to extract and analyze the implementation patterns of security-critical fields. This process consists of three main stages: security field analysis, middle summary, and meta summary, ultimately producing a finite and analyzable set of representative implementations that guide the development of our static detection tool. All prompts used in this study are publicly available in our open-source repository.

Security Fields Analysis. As demonstrated in Section VI, most state-of-the-art LLMs are capable of extracting verification function (VF) code snippets with high precision, achieving over 80% accuracy even in the least effective model. Based on this capability, we leverage an LLM to identify field-level patterns in real-world VFs. For each VF group identified during the grouping phase, we randomly select one representative function. The LLM, acting as an automated contract auditing expert, examines whether the target security field is present and correctly implemented.

To improve clarity, we format prompts using Markdown, following best practices from prior work [27], [28], [39], [41], [52], [53]. The System Prompt defines the LLM's role and outlines its tasks: checking the presence, correctness, and implementation of the field. The User Prompt provides task-specific details, including the target field's security definition, the required checks, and the expected JSON output format. To support accurate reasoning, the prompt includes not only the core VF but also relevant state variable declarations and external function calls. Finally, we aggregate all valid results for subsequent summarization and exclude samples in which the target field is absent.

Middle Summary. Using an LLM to summarize a large number of results presents two key challenges. First, the limited context length of the LLM prevents us from feeding all sample code into a single pass for summarization. Second, giving too much input at once can lead the model to overlook rare but valid implementation patterns.

To address challenges, we introduce a *batch-wise recursive summarization* approach, called the *middle summary*. Structured outputs from the previous stage serve as inputs for each round, where inputs are grouped into batches and summarized by the LLM. The resulting summaries are then used as input for the next round. If the number of summaries still exceeds the batch size or a predefined threshold, the process repeats. This method compresses the input logarithmically. Given N results and batch size b, the number of rounds is bounded by $O(\log_b N)$. For example, with $N = 10,000$ and $b = 10$, only about four rounds are needed. The total number of LLM requests is approximately $N/(b-1)$.

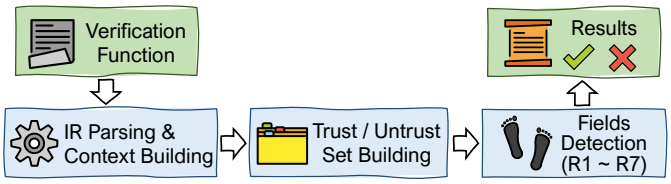

Fig. 4: The Workflow of VCSCOPE

The prompt structure follows that of the earlier analysis step but shifts focus to identifying and generalizing implementation patterns. Our prompt also instructs the LLM to ignore surface-level differences such as variable names and formatting, encouraging abstraction based on logic and structure.

Meta Summary. After the middle summary phase, we perform a final summarization called the *meta summary*. The LLM generates a report describing each representative pattern, providing example code snippets, and explaining its strengths or weaknesses. These results directly inform rule-based detection tools by showing how each field is implemented in practice. Specifically, the LLM distills common implementation paradigms from the aggregated batch summaries. The prompt for this stage extends the middle summary template and adds three guiding questions: (1) Can multiple implementations be abstracted into a unified pattern? (2) If correct, what are the security advantages of the implementation? (3) If incorrect, what causes the implementation to fail?

Implementation Patterns. From the summary results, the LLM identified diverse implementation patterns for each security-critical field. Specifically, the LLM extracted 17 patterns for `Signer`, 22 for `Nonce`, 12 for `ChainID`, and 12 for `VC address`. Further details are provided in Section VI. We manually reviewed all outputs, focusing on implementation diversity rather than LLM accuracy, and consolidated semantically similar implementations into a smaller set of representative patterns for rule-based detection.

V. VCSCOPE: A FINE-GRAINED DETECTION TOOL

Building on the LLM-summarized field patterns, we derive a concise set of detection rules that ensure comprehensive coverage across diverse implementations. This section presents our fine-grained detection tool, VCSCOPE. Section V-A details its architecture, while Section V-B elaborates on its detection methodology and rule design.

A. Architecture Overview

VCSCOPE (Verifier Contract Scope) is a static analysis tool that leverages Slither's [24] Intermediate Representation (IR) to detect the implementation of critical security fields in signature verification. As shown in Figure 4, for a given verification function (VF), VCSCOPE first parses the contract into IR and constructs its contextual representation. This includes identifying functions along the call chain, variable read/write sets, control-flow conditions, and parameter-passing

relationships. Using this context, VCSCOPE constructs two variable sets: a *trusted set* (e.g., state variables, blockchain variables) and an *untrusted set* (e.g., user-supplied inputs). These concepts will later be formalized in Section V-B. It then performs backward tracking from critical cryptographic primitives such as `ecrecover`, analyzing related conditional checks and assignments to iteratively update the trusted set. Finally, the tool applies predefined Rules 1–7 (Table I) to validate the correctness of the four security fields and outputs the detection results.

As a fine-grained tool, VCSCOPE thoroughly accounts for Solidity-specific features during context building. It employs extraction and expansion strategies to handle *inline assembly*, *structs*, *arrays*, and other constructs, and incrementally updates the call graph when constructor-initialized variables introduce new dependencies, ensuring comprehensive analysis coverage.

For AS1 and AS2, the tool analyzes all functions declared with `public` or `external` visibility as potential external entry points. For AS3, functions marked as `view` or `pure` are further examined if they return boolean values, as such functions are typically involved in off-chain verification. `View` functions can read but not modify contract state, allowing relaxed `Nonce` requirements, while `pure` functions cannot access any state and are directly flagged as potential risks.

B. Detection Methodology and Rule Design

This subsection formalizes the detection process by modeling verification functions (VFs), defining trusted and untrusted variables, and specifying how detection rules ensure the correctness of security field implementations.

Detection Methodology. We propose to assess the trustworthiness of variables in VFs to determine whether security-critical fields are correctly implemented. Variables are categorized into two disjoint sets: the trusted set $\mathcal{T}$ and the untrusted set $\mathcal{U}$. The trusted set $\mathcal{T}$ contains variables defined internally by the VF and cannot be controlled by users. In contrast, variables in $\mathcal{U}$ are considered untrusted, as they may be controlled by users (attackers). We define a set of detection rules to identify potential vulnerabilities. These rules examine how predicates in the VF constrain untrusted variables using trusted values. A secure VF must verify all security-critical variables, either through cryptographic bindings or conditional checks. If such checks are missing or insufficient, the function is marked as potentially vulnerable.

Next, we formalize the structure of verification functions and define related concepts such as cryptographic primitives and conditional constraints.

Verification Function. Let $\mathcal{VC}$ be a verifier contract, and let $\mathcal{VF}$ denote its signature VF. We model $\mathcal{VF}$ as an abstract semantic triple: $\mathcal{VF} = (\mathcal{I}, \mathcal{C}, \mathcal{S})$, where:

- $\mathcal{I}$ is the set of input parameters explicitly passed to $\mathcal{VF}$. These parameters are user-controlled, so they are considered untrusted variables, denoted as $\mathcal{I} \subseteq \mathcal{U}$.
- $\mathcal{C}$ is the set of contextual variables that $\mathcal{VF}$ depends on, including: 1) Contract state variables; 2) Blockchain global variables (e.g., `block.chainid`); 3) Contract metadata (e.g., `address(this)`). These variables are considered trusted, denoted as $\mathcal{C} \subseteq \mathcal{T}$.
- $\mathcal{S}$ is the set of expression statements within $\mathcal{VF}$, encompassing cryptographic primitives, conditional statements, and assignments.

Cryptographic Primitives. The $\mathcal{VF}$ relies on two essential cryptographic primitives from $\mathcal{S}$ to verify external inputs:

- The *hash function* $\mathcal{H}$ maps a structured message $\mathcal{M} = (x_1, x_2, \dots)$ to a fixed-length digest $h = \mathcal{H}(\mathcal{M})$. In Solidity, $\mathcal{H}$ is typically instantiated as `keccak256`, and h is a 256-bit digest.
- The *signature recovery function* $\mathcal{R}$ takes as input a message hash h and a signature `sig`. It returns the Ethereum address a that produced the signature if it is valid, and the zero address `0x0` otherwise. In Solidity, this primitive is provided by the built-in `ecrecover` function.

Conditional Statements. The $\mathcal{VF}$ typically performs signature verification using conditional statements, commonly implemented with `if` or `require` in Solidity. These conditional statements correspond to a set of Boolean predicates $\mathcal{P}$ contained in the statement set $\mathcal{S}$. Each predicate $\varphi \in \mathcal{P}$ imposes constraints over variables drawn from $\mathcal{I} \cup \mathcal{C}$, where $\mathcal{I} \subseteq \mathcal{U}$ (untrusted) and $\mathcal{C} \subseteq \mathcal{T}$ (trusted).

Trust Propagation. During the analysis, we continuously update the trusted set $\mathcal{T}$ to include variables that have been properly verified. We define a trust propagation mechanism that expands $\mathcal{T}$ based on cryptographic guarantees and program semantics. First, if the output of a cryptographic operation is trusted, its inputs can also be regarded as trusted due to the one-way and collision-resistant properties of hash functions. Second, trust can propagate through assignments: if a variable is directly assigned from a trusted one (e.g., $x = y$ with $y \in \mathcal{T}$), then x is added to $\mathcal{T}$.

Detection Rules. Table I summarizes our detection rules for identifying vulnerabilities, which are derived from implementation patterns identified in Section IV. The detection of `Signer` is fundamental, as it relies on the message and signature, which contain all critical fields. If `Signer` verification fails, checks on the other fields lose their meaning. Thus, rules 1 and 2 focus on `Signer` verification, while rules 3 to 7 build on that foundation to validate the remaining fields.

Signer Detection. To ensure `Signer` integrity, the $\mathcal{VF}$ must include a condition $\varphi \in \mathcal{P}$ that checks the validity of the recovered address $signer = \mathcal{R}(h, \texttt{sig})$. Based on the behavior of `ecrecover`, we define two detection rules:

- **R1: Trusted Address Check.** The condition φ ensures that $signer$ equals a trusted address a, and a is a state variable.
- **R2: Non-Zero Address Check.** The condition φ ensures $signer \neq$ `0x0`, indicating successful signature recovery.

Satisfying either rule is sufficient for considering the `Signer` valid. With trust propagation, a trusted $signer$ also implies trust in h, `sig`, and the message $\mathcal{M}$ used to compute h.

TABLE I: Detection Rules for Verification Functions

Field	Rule	Formal Definition
Signer	**R1**: Trusted Address Check	$signer = \mathcal{R}(h, \texttt{sig}),\ \exists\ \varphi \in \mathcal{P} : signer == a,\ a \in \mathcal{C} \subseteq \mathcal{T}$
	R2: Non-Zero Address Check	$signer = \mathcal{R}(h, \texttt{sig}),\ \exists\ \varphi \in \mathcal{P} : signer \neq \texttt{0x0}$
Nonce	**R3**: State-Based Nonce Check	$\exists\ nonce \in \mathcal{M} \cap \mathcal{C},\ \exists s \in \mathcal{S} : s \equiv nonce := \_,$
	R4: Input-Based Nonce Check	$\exists\ nonce \in \mathcal{C},\ \exists\ nonce' \in \mathcal{M} \cap \mathcal{I},\ \exists s \in \mathcal{S} : s \equiv nonce := \_$ $\exists\ \varphi \in \mathcal{P} : nonce == nonce' \vee nonce[\_] == nonce',$
	R5: Sig-Based Nonce Check	$\exists nonce \in \mathcal{C},\ \exists \varphi \in \mathcal{P} : nonce[\texttt{sig}] == \texttt{false},$ $\exists s \in \mathcal{S} : s \equiv nonce[\texttt{sig}] := \texttt{true},\ \exists \varphi' \in \mathcal{P} : \texttt{sig}_s \leq \texttt{SECP256K1N}/2$
Chain ID	**R6**: Blockchain Check	$\exists\ \varphi \in \mathcal{P} :\ \exists\ x \in \mathcal{M} : x == \texttt{block.chainId}$
VC Address	**R7**: Contract Address Check	$\exists\ \varphi \in \mathcal{P} :\ \exists\ x \in \mathcal{M} : x == \texttt{address(this)}$

Nonce Detection. As shown in Listing 2, the `Nonce` field may take different forms, but secure usage must satisfy three core properties: it must be stored in state ($nonce \in \mathcal{C}$), updated after use ($\exists\, s \in \mathcal{S} : s \equiv nonce := \_$), and included in the signed message ($nonce \in \mathcal{M}$). We define three detection rules:

- **R3: State-Based Nonce Check.** The message directly includes a state variable ($nonce \in \mathcal{M} \cap \mathcal{C}$). In this case, `Signer` verification (Rule 1 or 2) usually ensures `Nonce` correctness without requiring an explicit condition.
- **R4: Input-Based Nonce Check.** The nonce is given as input and included in the message ($nonce' \in \mathcal{M} \cap \mathcal{I}$). The contract must check that $nonce'$ matches the stored value, typically via $nonce == nonce'$. For per-user mappings, the condition becomes $nonce[\_] == nonce'$.
- **R5: Sig-Based Nonce Check.** The nonce is derived from the signature. The contract must ensure it has not been used by checking $nonce[\texttt{sig}] == \texttt{false}$, then marking it as used. To avoid malleability, the signature must also satisfy $\texttt{sig}_s \leq \texttt{SECP256K1N}/2$.

Chain ID and VC Address Detection. The $\mathcal{VF}$ must ensure the message is bound to the correct execution context, including the current blockchain and contract address.

- **R6: Blockchain Check.** This rule identifies conditions that explicitly compare a message field $x \in \mathcal{M}$ with the current blockchain, i.e., $x == \texttt{block.chainId}$.
- **R7: Contract Address Check.** This rule detects comparisons between a message field $x \in \mathcal{M}$ and the current `VC` address, i.e., $x == \texttt{address(this)}$.

In practice, `ChainID` and `VC address` are sometimes hard-coded in the message rather than provided as inputs. This is similar to Rule 3, where the message $\mathcal{M}$ directly includes the relevant values. As a result, we do not define separate rules for these cases in Table I.

VI. Evaluation

This section evaluates our approach. First, we assess the effectiveness of our LLM-based method. We then analyze the implementation of four security-critical fields across 22,374 real-world verification functions (VFs). Finally, we compare our rule-based detection tool with an LLM-based approach. Specifically, we address the three research questions:

TABLE II: LLM Performance Comparison

Model	R1	R2	R3	Mean	Std
DeepSeek-V3	0.860	0.850	0.860	0.857	0.0047
DeepSeek-R1	0.860	0.880	0.850	0.863	0.0125
Gemini-2.5-pro	0.850	0.830	0.810	0.830	0.0163
o4-mini-high	0.850	0.840	0.850	0.847	0.0047
GPT-5	0.820	0.800	0.810	0.810	0.0082

Note: R1–R3 denote accuracy in three independent testing rounds.

- **RQ1:** Does the LLM-based pattern mining approach effectively capture representative patterns?
- **RQ2:** Do real-world verification functions correctly implement all four security-critical fields?
- **RQ3:** How does the rule-based tool compare to the LLM-based approach in accuracy and robustness?

Dataset. We constructed our dataset based on the CryptoScan project [50], which includes 22,374 externally callable functions (`public/external`) from 14,581 smart contracts. Among them, 2,795 functions are read-only (`view/pure`).

Setup. Experiments were conducted on a server running Ubuntu 22.04.5 LTS with Python 3.12.10, two AMD EPYC 9654 processors (384 cores), and 1 TB of RAM. We employed the *DeepSeek-V3-0324* [5] model as the LLM due to its overall performance (Section VI-A). Each field was analyzed independently to avoid cross-sample interference. Following existing studies [25], [48], we set the temperature to 0.7 to balance exploration and stability. The tool's deployment is not dependent on any specific model, and with eight parallel threads, the entire experiment finished in under one hour.

A. RQ1: Effectiveness of LLM-Based Pattern Mining

To answer RQ1, we evaluate our approach from three complementary perspectives: (1) the capability of LLMs to perform pattern mining; (2) the coverage and consistency of two-stage summarization; and (3) the efficiency gain achieved through the verification function (VF) grouping.

Capability of LLMs. We first investigate whether state-of-the-art LLMs can accurately extract implementation patterns.

We selected five representative models, including *DeepSeek-V3*, *DeepSeek-R1*, *Gemini-2.5-Pro*, *o4-mini-high*, and *GPT-5*, based on the benchmark results of the public LLM leaderboard [3]. We then randomly sampled 100 VFs and executed each model in three independent runs, manually inspecting the correctness of the outputs to assess their capability.

As shown in Table II, all models achieved accuracy above 81%, indicating that state-of-the-art LLMs can reliably capture the implementation logic of security-critical fields. *DeepSeek-R1* achieved the best overall performance, yet it suffered from slow responses due to its extended reasoning phase and incurred higher token costs. Therefore, we employed *DeepSeek-V3* in the subsequent experiments, which provides comparable accuracy with faster responses and lower costs.

```
//Case 1 (Correct, 101 VFs): In-Place Incremented Nonce
mapping(address => uint256) public nonces;
function Nonce1(address _owner, ...) public {
    bytes32 digest = keccak256(abi.encodePacked(
                                nonces[_owner]++, ...))
    ...
}

//Case 2 (Correct, 13 VFs): Modifier-Based Nonce Management
uint256 public globalNonces;
function Nonce2(uint _nonce, ...) external useNonce(_nonce)
{...}
modifier useNonce(uint _nonce) {
    require(globalNonces++ == _nonce, "Inv");
}

//Case 3 (Correct, 124 VFs): Signature-Based Nonce
mapping(bytes => bool) public signatureUsed;
function Nonce3(..., bytes memory _sig) external {
    require(!signatureUsed[_sig], "used");
    require(recover(..., _sig), "Inv");
    require(uint256(_sig.s) <= 0x7FFF...20A0, "Inv")
    signatureUsed[_sig] = true;
    ...
}

//Case 4 (Incorrect, 50 VFs): Ineffective Nonce
address public signer; // a global signer
function Nonce4(..., uint nonce, bytes memory sig) external{
    require(getSigner(..., nonce, sig) == signer, "Inv");
    ...
}
```

Listing 2: Examples of `Nonce` Usage Patterns.

Two-Stage LLM Summarization. To assess the effectiveness of our two-stage summarization, we conducted both coverage and consistency evaluations on the generated meta-patterns. In a random sample of 100 VFs, every function was covered by at least one summarized pattern, confirming comprehensive coverage of both common and edge-case implementations. Repeated summarization produced consistent results, with the number of extracted patterns per field varying by fewer than three across trials. These minor variations are primarily due to differences in the granularity of LLM categorization, which do not affect the eventual rule derivation, as the resulting variants are semantically equivalent.

Listing 2 presents several representative `Nonce` handling patterns discovered in real-world VFs. Compared with prior experience-based studies [26], [38], our LLM-based analysis uncovered a broader range of implementation patterns. Case 1 (101 VFs) represents the most typical design, where a per-user `Nonce` is incremented in place and tightly bound to the signer. Case 2 (13 VFs) maintains a global counter through a modifier; Case 3 (124 VFs) adopts a signature-based scheme that enforces uniqueness by marking used signatures. Case 4 (50 VFs) reflects an ineffective pattern in which the `Nonce` is included but never validated.

TABLE III: Security Field Implementation Statistics in VFs

Field	AS1, AS2 (On-chain)	AS3 (Off-chain)
Signer	80.5% (15,762)	73.7% (868)
Nonce	41.0% (8,022)	23.8% (280)
ChainID	33.4% (6,533)	24.4% (287)
VC Address	59.1% (11,562)	39.1% (460)
Total VFs	19,579	1,177

Note that in this step, our goal is to use the LLM to capture diverse implementation patterns. Occasionally, the LLM may misclassify certain patterns, for example, treating an incorrect implementation as a correct one. However, such misclassifications do not affect rule extraction, as we manually refine these cases during rule construction. A detailed comparison of the LLM's detection capability is provided in RQ3 (Section VI-C).

Efficiency of Verification Function Grouping. We evaluated our VF grouping strategy on a dataset of 22,374 real-world VFs. After applying structural fingerprinting, the number of unique VF clusters was reduced to 5,404, representing 24.2% of the original dataset. Since each field analysis requires both code and prompt tokens, token consumption scales linearly with the number of analyzed functions. Without grouping, the total token usage reached 29.65 million. By analyzing only representative samples from each group, this number dropped to 7.62 million, yielding a 74.3% reduction.

B. RQ2: Security-Critical Fields in VFs

Table III presents the detection results for VFs. Among the 22,374 VFs analyzed, 19,579 are marked as `public` or `external`, which are relevant to AS1 and AS2. Additionally, 2,795 are *read-only* functions, related to AS3. Of these, 1,177 are `view` functions returning a boolean value and were subject to further analysis. We also identified 846 `pure` functions, including 223 that return a boolean. The `pure` functions cannot access state variables, so we flag these as security risks. Next, we begin by analyzing the field usage in on-chain scenarios, followed by those related to off-chain scenarios.

On-chain AS. Among the VFs used in on-chain scenarios, about 80.5% include the `Signer`, but other critical fields are often missing: Only 41.0% include the `Nonce`, and just 33.4% make use of the `ChainID`. The absence of security-critical fields does not immediately indicate a vulnerability, as business-related code often includes additional checks. For example, the `transfer` function in Listing 1 verifies whether the `from` address is the actual token owner (line 20). However,

the best practice is to ensure signature validity within the verification logic itself, rather than relying on specific business logic. We provide a detailed analysis of each field below.

Signer. According to Rule 1, the recovered address should be compared to a trusted state variable or at least checked against the zero address. However, we observe that some developers only compare the recovered address with an input parameter (referred to as `local_signer`). This allows an attacker to craft a malformed signature that causes `ecrecover` to return `0x0`, then set `local_signer` to `0x0`, thereby passing the `signer == local_signer` check. This pattern matches the typical bypass described in AS1. We found that some VFs are protected by mechanisms such as `onlyOwner`, which restrict access from regular users. However, as noted earlier, VFs should still perform complete signature verification.

Nonce. Our analysis reveals that many VFs either omit `Nonce` usage entirely or contain flawed implementations. Only 41.0% handle `Nonces` properly. A common flaw is using a message hash as the `Nonce` without marking it as *used*, failing to prevent replays. Some VFs use signatures as `Nonces` but do not verify the s value of each signature, as highlighted in Rule 5. Due to ECDSA malleability, this oversight allows an attacker to forge a different signature to bypass the nonce check.

Chain ID. `ChainID` shows the lowest correct implementation rate at only 33.4%. This indicates that many contracts fail to consider the risk of cross-chain replay attacks. Some VFs hardcode the `ChainID` using a fixed integer value, which violates best practices. If the contract is redeployed to another chain, developers may forget to update the hardcoded value, leading to potential cross-chain replay vulnerabilities.

VC Address. Over 40% of VFs fail to implement the VC address correctly. As discussed in Section III-B, this omission not only exposes the contract to cross-contract replay attacks, but also enables adversaries to deploy a clone contract that mimics the original VC. Such clones can be used to proactively initiate forged interactions, posing a serious threat to the integrity of the verification process.

Off-chain AS. The correct implementation rate of security-critical fields in off-chain (`view`) VFs is significantly lower compared to their on-chain counterparts. This suggests that many developers do not apply the same level of security rigor when designing `view` VFs, thereby introducing the risk of AS3. We also identified 223 `pure` VFs that exhibit AS3-related security risks. Since `pure` functions cannot access contract state, they are unsuitable for off-chain pre-verification. These functions were mainly intended for on-chain use but were mistakenly labeled as `public` or `external`, exposing them to unintended access. To prevent misuse, such functions should be marked as `private` or `internal`, restricting external calls.

C. RQ3: Rule-Based vs. LLM-Based Detection

This section compares the detection performance of our rule-based static analysis tool with that of a pure LLM-based detection approach. As introduced in Section IV-C, the LLM is directly prompted to evaluate the correctness of security-critical fields. Therefore, we compare its results with those obtained from our static analysis tool.

TABLE IV: Detection Performance of VCSCOPE vs. LLM

Field		Precision	Recall	F1-score	FP	FN
Signer	LLM	89.1%	93.2%	91.1%	10	6
	Tool	**100.0%**	**96.6%**	**98.3%**	0	3
Nonce	LLM	86.2%	59.5%	70.3%	4	17
	Tool	**96.6%**	**77.8%**	**86.4%**	1	8
ChainID	LLM	78.6%	47.8%	59.6%	3	12
	Tool	**100.0%**	**100.0%**	**100.0%**	0	0
VC Addr.	LLM	95.5%	55.3%	69.7%	1	17
	Tool	95.5%	**100.0%**	**97.7%**	1	0
Total	LLM	88.5%	72.8%	80.0%	18	52
	Tool	**98.7%**	**93.1%**	**95.8%**	2	11

We randomly selected 100 VFs. Two researchers independently reviewed the detection results and resolved inconsistencies through discussion. The final results are shown in Table IV. Overall, VCSCOPE achieves superior performance across all metrics, with an F1-score of 95.8%, significantly outperforming the LLM baseline, which achieves 80.0%. The performance gap is particularly pronounced in fields such as `ChainID` and `VC address`. To better understand this discrepancy, we conducted an in-depth analysis of both approaches and examined their respective strengths and weaknesses.

Function Call Resolution. Our tool can accurately resolve deeply nested function calls and track the propagation of security-critical fields. For example, it reliably identifies how variables such as `ChainID` and `VC address` are passed through multiple intermediate functions. In contrast, the LLM often fails to trace these intermediate calls, leading to incorrect detections and false positives.

Misleading Variable Names. The LLM heavily relies on variable names and tends to assume that variables with security-related names, such as `nonce`, correctly implement the intended security logic without further verification. This superficial reliance often leads to false positives when the actual validation logic is missing. In contrast, VCSCOPE identifies variable semantics based on predefined behavioral rules rather than names, effectively avoiding such misinterpretations.

Implicit Checks. Our rule-based detection approach can effectively handle implicit validation cases. For example, a check on one variable (e.g., `Signer`) may implicitly ensure the validity of another (e.g., `Nonce`). In contrast, the LLM often overlooks such implicit logic, leading to false negatives.

Semantic Understanding and Inference. Developers sometimes assign semantic roles to specific addresses, such as signer, executor, or transmitter, based on contextual logic. In such cases, our tool may fail to recognize these associations because they cannot be defined through a general rule. By leveraging its semantic reasoning capability, the LLM can often infer the intended role of certain variables, particularly those like `signer`.

VII. Discussion

This section discusses the limitations of our detection framework and the potential threats to its validity. We first present the inherent limitations of our detection tool, then analyze validity threats from two perspectives: the reliability of LLM-based detection and the representativeness of the dataset.

Limitation. This work focuses on four security-critical fields that define the minimal security boundary of the OSOV (Off-chain Signing and On-chain Verification) workflow. However, our detection does not account for additional checks embedded in the business-logic portion of verification functions. In some implementations, verification functions are protected by modifiers such as `onlyOwner` or by other context-specific constraints, which are beyond the scope of our current detection tool. However, we believe that a verifier contract should independently validate these critical fields to ensure the integrity and consistency of signature verification, rather than relying on specific business logic.

Threats to Validity. The reliability of LLM-based detection represents a potential threat to the validity of our results. LLM performance may vary across model versions, training data, or fine-tuning alignment, which can affect both reasoning quality and output consistency. Moreover, LLM outputs are inherently sensitive to prompt design and temperature settings. In our evaluation, all models were tested using the same set of prompts, and the temperature was selected based on prior studies and empirical experience. However, it is possible that certain LLMs could achieve better performance under refined prompts or different parameter configurations.

Another potential threat arises from the dataset. Our dataset primarily collected from Ethereum, which may limit the generalizability of our findings to other ecosystems. In principle, the OSOV workflow and the definitions of security-critical fields are applicable across different platforms. In addition, we only included contracts with available Solidity source code. This restriction may introduce a bias in the mined patterns, as bytecode-only contracts might follow different design conventions that are not reflected in our dataset.

VIII. Related Work

This section reviews existing research on signature vulnerabilities, static analysis techniques for smart contracts, and the emerging use of LLMs in vulnerability detection.

Existing studies have explored security risks from multiple perspectives in signature vulnerability detection, including the misuse of off-chain signing methods, flaws in on-chain verification logic, and weaknesses in cryptographic implementations. SigScope [35] was the first to systematically analyze the security risks of off-chain message signing in decentralized applications, revealing several typical types of signature-related vulnerabilities. Web3AuthChecker [47] proposed a dynamic analysis tool to detect blind message attacks in Web3 authentication processes, also focusing on off-chain signing scenarios. CryptoScan [50] analyzed a large number of real-world security audit reports and defined twelve common categories of cryptographic defects in smart contracts. Siguard [51] applied symbolic execution at the bytecode level to identify security flaws in verification logic, with particular attention to missing contract addresses or storage variables. Beyond these studies focusing on signing and cryptographic mechanisms, researchers have developed a range of static analysis tools for smart contract vulnerability detection, such as Eth2vec [20], eThor [37], Securify [43], SmartCheck [42], and Naga [46]. These tools detect various classes of security flaws, including reentrancy, access control, and other common vulnerabilities.

Unlike prior studies, our work takes a holistic view of the OSOV workflow and defines its minimal security boundary. We identify four critical security fields that must be properly validated in verification functions. Whereas existing research focuses on specific vulnerabilities, our work emphasizes the correctness of the verification logic itself, representing a different research perspective. Rather than depending on manually crafted rules or audit reports, our method employs an LLM to extract implementation patterns of key security fields, which then guide rule construction and improve coverage.

Recent studies have demonstrated the potential of LLMs in vulnerability detection [40], [30], [44], automated repair [22], and security suggestion generation [34], [33]. LLMs possess strong semantic understanding and contextual reasoning abilities, which enable them to capture boundary logic that traditional static analysis tools may miss. However, directly applying LLMs to vulnerability detection remains challenging due to difficulties in tracking inter-procedural calls and handling complex data-flow dependencies. Therefore, instead of using LLMs for direct detection, our work leverages their strength to summarize implementation patterns, which then guide the design of our rule-based detection tool.

IX. Conclusion

This paper systematically examines the security of Off-chain Signing and On-chain Verification (OSOV) and defines its minimal security boundary through four essential fields. By leveraging LLMs, we effectively uncover diverse implementation variants of these fields across real-world verifier contracts, which in turn guide the design of our fine-grained static detection tool, VCScope. Experimental results demonstrate both the capability of LLMs and the effectiveness of our detection tool, and further reveal that a significant portion of existing verifier contracts fail to meet these basic security requirements. While LLMs are still limited as standalone detectors, their semantic understanding effectively complements rule-based analysis. Future work can explore hybrid frameworks for more adaptive and comprehensive security auditing.

Data Availability

All research artifacts, including the LLM prompts for pattern mining (Section IV), the static analysis tool VCScope (Section V), and the datasets used in the experiments (Section VI), are available at https://github.com/w030w/VCScope.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments. This work was partially supported Natural Science Foundation of Shandong Province (Grant No. ZR2023MF043) and Taishan Young Scholar Program of Shandong Province, China (Grant No. tsqn202211001).

REFERENCES

[1] OpenSea Now Has a Valuation of $13.3 Billion. https://coinmarketcap.com/academy/article/opensea-now-has-a-valuation-of-13-3-billion, 2022. (Accessed on 10/10/2025).

[2] Decentralized exchange volume hits record high of $462B in December. https://cointelegraph.com/news/dex-all-time-high-monthly-volume-462-billion, 2024. (Accessed on 10/10/2025).

[3] Artificial Analysis. https://artificialanalysis.ai/, 2025. (Accessed on 10/10/2025).

[4] Compound Governance. https://compound.finance/governance/comp, 2025. (Accessed on 10/10/2025).

[5] DeepSeek. https://www.deepseek.com/, 2025. (Accessed on 10/10/2025).

[6] EIP-2: Homestead Hard-fork Changes. https://eips.ethereum.org/EIPS/eip-2, 2025. (Accessed on 10/10/2025).

[7] EIP-712: Typed structured data hashing and signing. https://eips.ethereum.org/EIPS/eip-712, 2025. (Accessed on 10/10/2025).

[8] ERC-1271: Standard Signature Validation Method for Contracts. https://eips.ethereum.org/EIPS/eip-1271, 2025. (Accessed on 10/10/2025).

[9] ERC-2612: Permit Extension for EIP-20 Signed Approvals. https://eips.ethereum.org/EIPS/eip-2612, 2025. (Accessed on 10/10/2025).

[10] ERC-4494: Permit for ERC-721 NFTs. https://eips.ethereum.org/EIPS/eip-4494, 2025. (Accessed on 10/10/2025).

[11] ERC-721: Non-Fungible Token Standard. https://eips.ethereum.org/EIPS/eip-721, 2025. (Accessed on 10/10/2025).

[12] ERC-7604: ERC-1155 Permit Approvals. https://eips.ethereum.org/EIPS/eip-7604, 2025. (Accessed on 10/10/2025).

[13] Ethereum: A Secure Decentralised Generalised Transaction Ledger. https://ethereum.org/content/developers/tutorials/yellow-paper-evm/yellow-paper-berlin.pdf, 2025. (Accessed on 10/10/2025).

[14] Opensea. https://opensea.io/, 2025. (Accessed on 10/10/2025).

[15] OpenZeppelin Governor Module. https://docs.openzeppelin.com/contracts/4.x/governance, 2025. (Accessed on 10/10/2025).

[16] Openzepplin. https://docs.openzeppelin.com/, 2025. (Accessed on 10/10/2025).

[17] Snapshot. https://snapshot.box/, 2025. (Accessed on 10/10/2025).

[18] uniswap. https://v4.uniswap.org/, 2025. (Accessed on 10/10/2025).

[19] Wyvern Protocol. https://wyvernprotocol.com/, 2025. (Accessed on 10/10/2025).

[20] Nami Ashizawa, Naoto Yanai, Jason Paul Cruz, and Shingo Okamura. Eth2Vec: Learning Contract-Wide Code Representations for Vulnerability Detection on Ethereum Smart Contracts. In *Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure, (BSCI 2021)*, 2021.

[21] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In *Proceedings of 2015 IEEE Symposium on Security and Privacy, (IEEE S&P 2015)*, 2015.

[22] Islem Bouzenia, Premkumar T. Devanbu, and Michael Pradel. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *Proceedings of 47th IEEE/ACM International Conference on Software Engineering, (ICSE 2025)*, 2025.

[23] Vitalik Buterin. A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper/, 2014. (Accessed on 10/10/2025).

[24] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, (WETSEB@ICSE 2019)*, 2019.

[25] Chang Gao, Haiyun Jiang, Deng Cai, Shuming Shi, and Wai Lam. StrategyLLM: Large Language Models as Strategy Generators, Executors, Optimizers, and Evaluators for Problem Solving. In *Proceedings of Neural Information Processing Systems 38, (NeurIPS 2024)*, 2024.

[26] Asem Ghaleb and Karthik Pattabiraman. How Effective are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools Using Bug Injection. In *Proceedings of 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, (ISSTA 2020)*, 2020.

[27] Zeyu He, Saniya Naphade, and Ting-Hao Kenneth Huang. Prompting in the Dark: Assessing Human Performance in Prompt Engineering for Data Labeling When Gold Labels Are Absent. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems, (CHI 2025)*, 2025.

[28] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.*, 2024.

[29] Kexin Hu and Zhenfeng Zhang. Fast Lottery-Based Micropayments for Decentralized Currencies. In *Proceedings of the 23rd Information Security and Privacy - Australasian Conference, (ACISP 2018)*, 2018.

[30] Zongwei Li, Xiaoqi Li, Wenkai Li, and Xin Wang. SCALM: Detecting Bad Practices in Smart Contracts Through LLMs. In *Proceedings of the AAAI Conference on Artificial Intelligence, (AAAI 2025)*, 2025.

[31] Zeqin Liao, Zibin Zheng, Xiao Chen, and Yuhong Nan. SmartDagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability. In *Proceedings of 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, (ISSTA 2025)*, 2022.

[32] Qi Lin, Binbin Gu, and Faisal Nawab. RollStore: Hybrid Onchain-Offchain Data Indexing for Blockchain Applications. *IEEE Trans. Knowl. Data Eng.*, 2024.

[33] Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. PropertyGPT: LLM-driven Formal Verification of Smart Contracts through Retrieval-Augmented Property Generation. In *Proceedings of 32nd Annual Network and Distributed System Security Symposium, (NDSS 2025)*, 2025.

[34] Wei Ma, Daoyuan Wu, Yuqiang Sun, Tianwen Wang, Shangqing Liu, Jian Zhang, Yue Xue, and Yang Liu. Combining Fine-Tuning and LLM-Based Agents for Intuitive Smart Contract Auditing with Justifications. In *Proceedings of 47th IEEE/ACM International Conference on Software Engineering, (ICSE 2025)*, 2025.

[35] Sajad Meisami, Hugo Dabadie, Song Li, Yuzhe Tang, and Yue Duan. SigScope: Detecting and Understanding Off-Chain Message Signing-related Vulnerabilities in Decentralized Applications. In *Proceedings of the ACM on Web Conference 2025, (WWW 2025)*, 2025.

[36] Daniel Perez and Benjamin Livshits. Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited. In *Proceedings of 30th USENIX Security Symposium, (USENIX Security 2021)*, 2021.

[37] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security, (CCS 2020)*, 2020.

[38] Tanusree Sharma, Kyrie Zhixuan Zhou, Andrew Miller, and Yang Wang. A Mixed-Methods Study of Security Practices of Smart Contract Developers. In *Proceedings of 32nd USENIX Security Symposium, (USENIX-Sec 2023)*, 2023.

[39] Hari Subramonyam, Divy Thakkar, Andrew Ku, Jürgen Dieber, and Anoop K. Sinha. Prototyping with Prompts: Emerging Approaches and Challenges in Generative AI Design for Collaborative Software Teams. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems, (CHI 2025)*, 2025.

[40] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, (ICSE 2024)*, 2024.

[41] Jiajin Tang, Ge Zheng, Jingyi Yu, and Sibei Yang. CoTDet: Affordance Knowledge Prompting for Task Driven Object Detection. In *Proceedings of IEEE/CVF International Conference on Computer Vision, (ICCV 2023)*, 2023.

[42] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *Proceedings of 1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, (WETSEB@ICSE 2018)*, 2018.

[43] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, (CCS 2018)*, 2018.

[44] Haijun Wang, Yurui Hu, Hao Wu, Dijun Liu, Chenyang Peng, Yin Wu, Ming Fan, and Ting Liu. Skyeye: Detecting Imminent Attacks via Analyzing Adversarial Smart Contracts. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, (ASE 2024)*, 2024.

[45] Libin Xia, Jiashuo Zhang, Che Wang, Zezhong Tan, Jianbo Gao, Zhi Guan, and Zhong Chen. Cryptcoder: An Automatic Code Generator for Cryptographic Tasks in Ethereum Smart Contracts. In *Proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering, (SANER 2024)*, 2024.

[46] Kailun Yan, Jilian Zhang, Xiangyu Liu, Wenrui Diao, and Shanqing Guo. Bad Apples: Understanding the Centralized Security Risks in Decentralized Ecosystems. In *Proceedings of the ACM Web Conference 2023, (WWW 2023)*, 2023.

[47] Kailun Yan, Xiaokuan Zhang, and Wenrui Diao. Stealing Trust: Unraveling Blind Message Attacks in Web3 Authentication. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security, (CCS 2024)*, 2024.

[48] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Proceedings of Neural Information Processing Systems 36, (NeurIPS 2023)*, 2023.

[49] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town Crier: An Authenticated Data Feed for Smart Contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, (CCS 2016)*, 2016.

[50] Jiashuo Zhang, Jiachi Chen, Yiming Shen, Tao Zhang, Yanlin Wang, Ting Chen, Jianbo Gao, and Zhong Chen. When Crypto Fails: Demystifying Cryptographic Defects in Ethereum Smart Contracts. *IEEE Trans. Software Eng.*, 2025.

[51] Jiashuo Zhang, Yue Li, Jianbo Gao, Zhi Guan, and Zhong Chen. Siguard: Detecting Signature-Related Vulnerabilities in Smart Contracts. In *Proceedings of 45th IEEE/ACM International Conference on Software Engineering: ICSE 2023 Companion Proceedings, (ICSE-Companion 2023)*, 2023.

[52] Tanghaoran Zhang, Yue Yu, Xinjun Mao, Shangwen Wang, Kang Yang, Yao Lu, Zhang Zhang, and Yuxin Zhao. Instruct or interact? exploring and eliciting llms' capability in code snippet adaptation through prompt engineering. In *Proceedings of 47th IEEE/ACM International Conference on Software Engineering, (ICSE 2025)*, 2025.

[53] Ruilin Zhao, Feng Zhao, Long Wang, Xianzhi Wang, and Guandong Xu. KG-CoT: Chain-of-Thought Prompting of Large Language Models over Knowledge Graphs for Knowledge-Aware Question Answering. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, (IJCAI 2024)*, 2024.

[54] Huijuan Zhu, Lei Yang, Liangmin Wang, and Victor S. Sheng. A Survey on Security Analysis Methods of Smart Contracts. *IEEE Trans. Serv. Comput.*, 2024.