TrapHunter: Exposing Covert Pathways in Trap Token Contracts

YIN WU, Xi'an Jiaotong University, China
YIXUAN LIU, Nanyang Technological University, Singapore
YI LI, Nanyang Technological University, Singapore
CHENYANG PENG, Xi'an Jiaotong University, China
HAO WU, Xi'an Jiaotong University, China
MING FAN, Xi'an Jiaotong University, China
TING LIU, Xi'an Jiaotong University, China
HAIJUN WANG*, Xi'an Jiaotong University, China

Standardized token contracts (e.g., ERC-20) form the foundation of digital assets. However, attackers increasingly abuse this standardization to disguise malicious trap tokens. Unlike obvious violations, these contracts employ a strategy of "deceptive adherence": they strictly adhere to standard protocols to evade detection, while embedding covert logic to defraud users. To address this, we first systematize the trap landscape by proposing a novel taxonomy derived from the intrinsic functional lifecycle of tokens (Generation, Circulation, Persistence, and Observation). We then propose TrapHunter, a framework designed to identify these traps and expose covert pathways within these deceptive contracts via intent deviation analysis. Specifically, TrapHunter introduces a unified semantic representation combining Abstract Behavior Trees (ABTs) and Augmented Path Graphs (APGs) to normalize intra-procedural syntax and reveal the hidden execution paths driven by inter-procedural state dependencies. Crucially, it bridges the semantic gap by leveraging LLMs to reason about the behavioral intent of deviations from reference implementations, followed by a fork-based dynamic validation to confirm exploitability. Experimental evaluations on 269 real-world contracts with three LLMs (DeepSeek, GPT, and Gemini) demonstrate that TrapHunter effectively detects all six categories of traps, achieving an average precision of 81.8% and recall of 85.4%, significantly outperforming state-of-the-art tools.

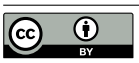
CCS Concepts: • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Trap tokens, Smart contract security, LLM-based analysis

ACM Reference Format:
Yin Wu, Yixuan Liu, Yi Li, Chenyang Peng, Hao Wu, Ming Fan, Ting Liu, and Haijun Wang. 2026. TrapHunter: Exposing Covert Pathways in Trap Token Contracts. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA015 (October 2026), 24 pages. https://doi.org/10.1145/3832106

*Corresponding author.

Authors' Contact Information: Yin Wu, Xi'an Jiaotong University, Xi'an, China, wuyin@stu.xjtu.edu.cn; Yixuan Liu, Nanyang Technological University, Singapore, Singapore, liuy0255@e.ntu.edu.sg; Yi Li, Nanyang Technological University, Singapore, Singapore, yi_li@ntu.edu.sg; Chenyang Peng, Xi'an Jiaotong University, Xi'an, China, pcy3123157003@stu.xjtu.edu.cn; Hao Wu, Xi'an Jiaotong University, Xi'an, China, emmanuel_wh@stu.xjtu.edu.cn; Ming Fan, Xi'an Jiaotong University, Xi'an, China, mingfan@mail.xjtu.edu.cn; Ting Liu, Xi'an Jiaotong University, Xi'an, China, tingliu@mail.xjtu.edu.cn; Haijun Wang, Xi'an Jiaotong University, Xi'an, China, haijunwang@xjtu.edu.cn.



ACM 2994-970X/2026/10-ARTISSTA015
https://doi.org/10.1145/3832106

1 Introduction

Blockchain technology has established a decentralized foundation for digital assets [6, 55, 59], with standardized token contracts (e.g., ERC-20) serving as the cornerstone of the Decentralized Finance (DeFi) ecosystem [52]. The widespread adoption of these standards is built on an implicit trust model: users and exchanges expect tokens to adhere to the behavioral baselines [60] established by widely used reference implementations (e.g., OpenZeppelin [31]). For instance, a `transfer` operation is expected to simply move assets, without hidden side effects.

However, this standardization has inadvertently lowered the barrier for a sophisticated class of fraud: **Trap Tokens**. These contracts strictly adhere to standardized protocols to pass basic verification checks but embed covert malicious logic within the implementation details [40]. The scale of this threat is significant: as of July 2025, Etherscan lists 1,646,089 token contracts, yet only 2,147 (0.2%) carry an "OK" or "Neutral" reputation [14], underscoring the dominance of suspicious or unverified tokens. Unlike conventional threats that rely on code vulnerabilities (e.g., reentrancy [48]) or crude financial scams [7, 44, 46, 51] (e.g., rug pulls via liquidity removal), trap tokens employ a strategy of deceptive adherence. Attackers embed malicious logic into implementation details through covert pathways, where execution flows are hijacked based on specific state conditions. A notorious example is the Squid Game (SQUID) token, which caused losses exceeding $3.3 million. While superficially compliant with standard interfaces, the contract embedded a hidden sell-restriction mechanism. This logic selectively blocked ordinary users from selling tokens to the liquidity pool while covertly whitelisting the developer's address, allowing only the attackers to drain the funds. Since such behaviors are syntactically correct and do not trigger runtime errors [58], they evade detection by traditional vulnerability scanners (e.g., Mythril, Slither) which focus on coding defects rather than malicious intent [47, 54].

Detecting trap tokens presents three fundamental challenges. **C1:** malicious logic is often deeply woven into complex inheritance hierarchies or disguised as benign administrative features [43], effectively burying the covert pathways amidst code and making them difficult to isolate. **C2:** a deviation from the standard implementation is not inherently malicious; legitimate tokens often customize logic for governance. Distinguishing a benign extension from a malicious trap requires reasoning about the *intent* [34] behind the code, a task where traditional symbolic execution struggles. **C3**: the malicious logic is embedded in source code, but traditional static analysis fails to capture the implicit cross-function dependency that connects the trigger condition (e.g., setBlacklist) to the trap behavior (e.g., transfer reversion).

To address these challenges, we propose TrapHunter, a novel intent deviation detection framework. TrapHunter operates on the premise that trap tokens can be identified by analyzing their semantic deviations from benign reference implementations and exposing hidden covert pathways. Specifically, to tackle the structural complexity of hidden pathways (**C1**), we first introduce a unified intermediate representation comprising Abstract Behavior Trees (ABTs) and Augmented Path Graphs (APGs). ABTs normalize intra-procedural logic to filter out syntactic noise, while APGs resolve complex inheritance structures and capture inter-procedural state dependencies (e.g., how a Solidity modifier reads a variable changed by another function). Based on this representation, TrapHunter employs a funnel-based detection workflow that bridges the gaps identified in **C2** and **C3**. It leverages Large Language Models (LLMs) [1, 19] to perform Path-Level Inconsistency Analysis (PIA), including DeepSeek-v3 [11], GPT-5 [30], and Gemini-2.5-pro [36]. Specifically, these models interpret the behavioral intent of semantic deviations to filter out benign customizations. To ensure reliability, it subsequently applies a fork-based behavioral validation, dynamically replaying suspicious paths on a reconstructed on-chain state to confirm exploitability.

We further conduct a comprehensive empirical study to systematize the landscape of trap tokens. By mapping malicious behaviors observed in real-world incidents to the intrinsic functional lifecycle of standardized tokens (Generation, Circulation, Persistence, and Observation), we derive a taxonomy of six distinct trap categories. Our evaluation on a curated dataset of 269 real-world contracts (comprising 501 labeled samples) demonstrates that TrapHunter effectively identifies traps, achieving an average precision of 81.8% and recall of 85.4% across three distinct LLMs.

In summary, this paper makes the following contributions:

- **Lifecycle-Based Taxonomy.** We propose a novel taxonomy of trap tokens derived from the intrinsic functional lifecycle of standardized tokens. By mapping malicious deviations to core phases(Generation, Circulation, Persistence, and Observation), we provide a theoretical basis for understanding how standardized protocols are weaponized.
- **Unified Semantic Representation.** We design a dual-layer representation framework combining ABTs and APGs. This approach effectively penetrates structural obfuscation by normalizing intra-procedural syntax and exposing the covert pathways driven by hidden inter-procedural state dependencies across inheritance hierarchies.
- **Intent-Aware Detection Framework.** We present TrapHunter, a framework that combines the semantic reasoning of LLMs with the rigorous verification of fork-based execution. This hybrid approach effectively bridges the semantic gap, distinguishing malicious traps from benign functional extensions while eliminating hallucinations.
- **Empirical Validation.** We curate a comprehensive dataset of 501 labeled samples sourced from verified real-world exploits and backdoor registries. Extensive evaluations demonstrate that TrapHunter outperforms state-of-the-art tools, achieving an average **precision of 81.8%, recall of 85.4%, and F1-score of 83.5%** across three distinct LLMs. Furthermore, we provide a detailed cost analysis and case study to demonstrate the system's practicality and robustness.

2 Background

2.1 Token Standards and Token Ecosystems

Tokens are blockchain-based digital assets, managed via smart contracts [53] and traded on platforms such as Decentralized Exchanges (DEXs) [51]. Smart contracts on Ethereum predominantly follow established token standards to ensure the operability of cryptocurrency exchanges [7]. For example, the ERC-20 [38] defines a standard interface for fungible tokens, specifying core behaviors [16] such as token generation (`mint`), value transfer (`transfer`), and approval mechanisms (`approve`). While the token standard defines interfaces, it does not mandate implementation details. Then the ecosystem heavily relies on the reference implementation provided by widely audited libraries like OpenZeppelin. This reference implementation establishes the implicit trust model for users and DEXs: users expect a transfer call to simply move assets without hidden side effects.

2.2 Malicious Tokens

Despite standardization, the ecosystem is plagued by financial fraud, generally categorized into three types. The first involves vulnerabilities caused by unintentional coding errors (e.g., reentrancy, integer overflows) that allow external attackers to drain funds [41]. The second comprises financial Scam tokens [37, 45, 51] where developers abscond with investors' funds. These include Rug-pulls [21, 51] (unilateral liquidity withdrawal), Ponzi schemes (redistributing capital from new to earlier investors), and traditional Honeypots [23, 37, 50] (as a decoy to lure *hackers* into exploiting a perceived vulnerability). Thirdly, we identify trap tokens. Instead of relying on exploitable bugs or obvious scams [5, 9], attackers deploy contracts that are syntactically compliant with standard interfaces but semantically malicious. Trap Tokens differ from the previous categories: (1) Unlike

traditional honeypots that target hackers, Trap Tokens are specifically crafted to deceive *regular investors*; (2) Unlike *Rug Pulls* which describe a financial outcome, Trap Tokens represent the *logic-based mechanism* (e.g., covert pathways) that facilitates the theft; and (3) Conceptually, they utilize *Backdoor techniques* [5, 28, 43]-such as privileged access control-but uniquely weaponize them within standard workflows (e.g., `transfer`) to lock in legitimate users under specific conditions.

3 Analysis of Trap Token Contracts

To systematize the trap token, we develop a taxonomy based on an empirical study of 269 real-world malicious smart contracts. We first describe the Open Card Sorting (OCS) employed for threat categorization and then organize these patterns according to the token's execution lifecycle.

3.1 Taxonomy Construction: Open Card Sorting

OCS [33] is a widely utilized method to establish an unbiased taxonomy, which allows categories to emerge inductively from the data rather than being imposed by pre-existing biases. Figure 1 shows a data card that contains three key pieces of information: codes, issues, and mechanism. In our study, a total of 269 cards were generated. *Issues* capture the external manifestation of a trap from a user's perspective (e.g., transaction reversion, failed selling). *Mechanism* provides a concise description of the internal state changes or control-flow dependencies causing the symptom.

Then the sorting process involved three researchers: two independent sorters (Ph.D. students with 2 years of auditing experience) and one senior arbiter (a security expert with >3 years of experience). The process was conducted in two rounds: **First**, sorters independently classified 40% of randomly selected cards (approx. 100 cards) and then grouped cards based on semantic similarity in their Underlying Logic and assigned descriptive labels to each group. **Second**, the sorters compared their groupings. Consistent groups (e.g., Blacklist vs. Blocklist) were merged. Disagreements where the same logic was categorized differently were resolved by the senior arbiter. The arbiter aligned the categories with the token lifecycle phases (discussed in Section 3.3) to ensure theoretical soundness. This process yielded six distinct trap categories. The remaining dataset was then labeled based on these finalized definitions.

3.2 Taxonomy Derivation: The Malicious Logic Lifecycle

To ensure the completeness of our taxonomy, we analyze the intrinsic functional lifecycle of standard tokens, which fundamentally defines four core state transitions: Generation, Circulation, Persistence, and Observation. We argue that Trap Tokens are covert semantic deviations injected into these lifecycle stages. By mapping malicious logic to these standard components, we derive a taxonomy that systematically covers the intent deviation attack of tokens as illustrated in Figure 2.

In the generation phase of the standard lifecycle, tokens enter circulation via minting. Attackers exploit this by embedding Infinite Mint (IM) logic, which allows privileged accounts to arbitrarily inflate the supply and dilute the value of legitimate holders. **In the circulation phase**, the core

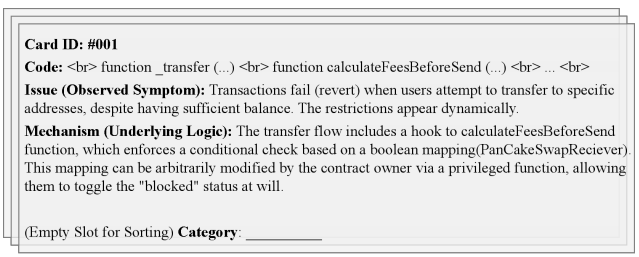


Fig. 1. Example of a card

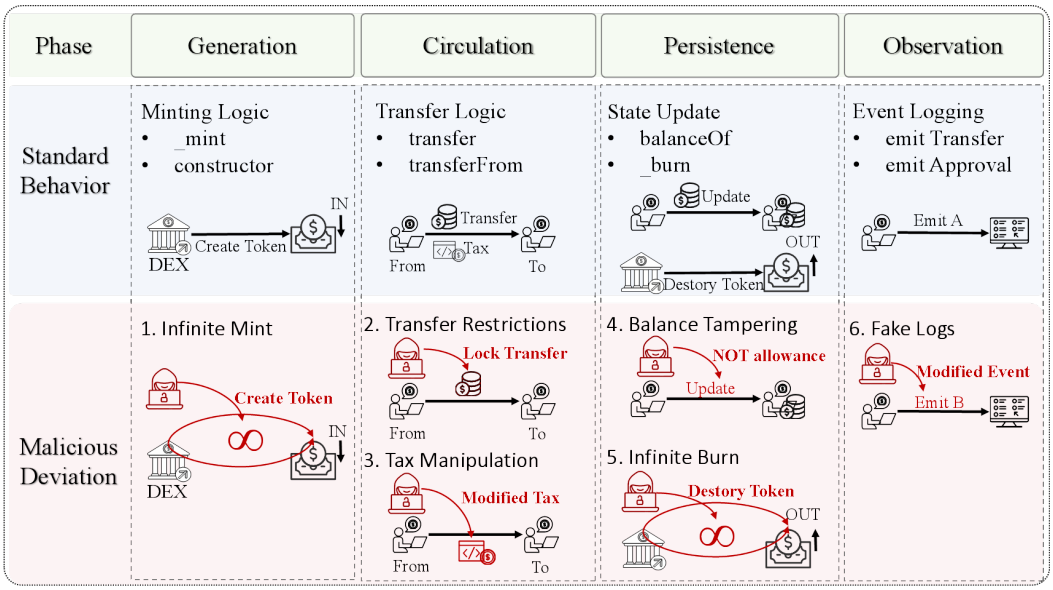


Fig. 2. Taxonomy of Trap Tokens mapped to the standard Functional Lifecycle

utility of a token lies in its transferability (`transfer`, `transferFrom`). This is the most heavily exploited stage. We identify two malicious deviations, Transfer Restrictions (TR), which selectively block sell orders to create "honeypot" effects, and Tax Manipulation (TM), which dynamically alters transaction fees to drain user funds during transfers. **In the persistence phase**, the standard guarantees that token balances (`balanceOf`) may change via authorized transfers or legitimate burning. Trap tokens compromise integrity through Balance Tampering (BT), which directly modifies storage slots to revoke user funds without requiring allowances, and Infinite Burn (IB), which allows attackers to destroy user tokens without proper authorization. **In the observation phase**, to maintain the facade of compliance, attackers manipulate the observation layer. Fake Logs (FL) emit standard-compliant events that contradict actual state changes, deceiving off-chain monitoring tools and users [24].

3.3 Definition of Trap Tokens.

Definition. A smart contract that maintains strictly **syntactic compliance** with the interface specifications of established token standards (e.g., ERC-20), while embedding **covert semantic deviations** within its implementation logic to defraud legitimate users.

Example. It is crucial to distinguish Trap Tokens from other defect types (as detailed in Section 2.2). Unlike vulnerabilities due to unintentional errors, Trap Tokens are engineered with deliberate malicious intent. Furthermore, unlike hacker-oriented honeypots, Trap Tokens target ordinary investors by weaponizing standard functionalities (e.g., `transfer`) to lock funds or manipulate balances. To illustrate how these traps manifest in practice, Figure 3 presents a code snippet of a *Transfer Restriction (TR)* trap, extracted from a confirmed scam contract in our ETH-BSC dataset. Superficially, the function calculateFeesBeforeSend (lines 17-23) appears to be a benign utility for calculating transaction fees as legitimate tokens. However, the attacker has embedded a hidden conditional branch (line 19) tied to a boolean mapping PanCakeSwapReciever. By toggling this variable via a privileged function (lines 25-28), the owner can silently activate a trap that causes user sell transactions to revert. This case exemplifies "deceptive adherence", where the code is syntactically valid but semantically malicious.

```solidity
function _transfer(address sender, address recipient, uint256 amount) internal virtual {
    require(sender != address(0), "ERC20:_transfer_from_the_zero_address");
    require(recipient != address(0), "ERC20:_transfer_to_the_zero_address");
    require(amount > 1000, "amount_too_small,_maths_will_break");
    _beforeTokenTransfer(sender, recipient, amount);
    _balances[sender] = _balances[sender].sub(amount, "ERC20:_transfer_amount_exceeds_balance");
    (uint256 transferToAmount, uint256 transferToFeeDistributorAmount) =
        calculateFeesBeforeSend(sender, recipient, amount);
    _balances[recipient] = _balances[recipient].add(transferToAmount);
    emit Transfer(sender, recipient, transferToAmount);
    if (transferToFeeDistributorAmount > 0 && feeDistributor != address(0)) {
        _balances[feeDistributor] = _balances[feeDistributor].add(transferToFeeDistributorAmount);
        emit Transfer(sender, feeDistributor, transferToFeeDistributorAmount);
    }
}

function calculateFeesBeforeSend(address sender, address recipient, uint256 amount) public view
     returns (uint256, uint256) {
    require(sender != address(0), "ERC20:_transfer_from_the_zero_address");
    if (PanCakeSwapReciever[recipient]) {
        revert("Error:_Can_not_sell_this_token");
    }
    return (amount, 0);
}

function setPanCakeSwapReciever(address _recipient, bool _feeless) public onlyOwner {
    require(_recipient != address(0), "ERC20:_transfer_from_the_zero_address");
    PanCakeSwapReciever[_recipient] = _feeless;
}
```

Fig. 3. Example of Transfer Restriction (TR) in a token contract.

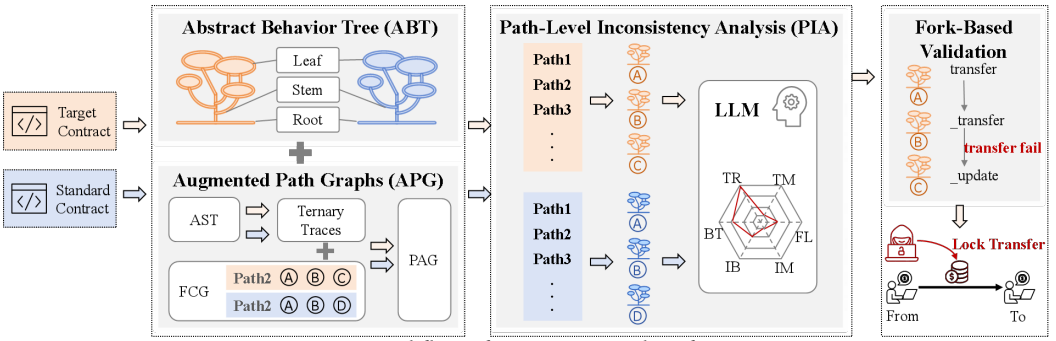


Fig. 4. Workflow of trap contract identification

4 Methodology

As illustrated in Figure 4, TrapHunter operates in three progressive stages: building a unified semantic representation using ABTs and APGs, reasoning about intent deviations via the LLM-powered PIA module, and dynamically replaying suspicious pathways under real-world on-chain states to confirm exploitability.

4.1 Abstract Behavior Tree (ABT)

To penetrate the syntactic disguise of trap tokens, TrapHunter structurally decomposes Solidity contracts to extract intra-procedural semantics. Unlike ASTs that retain redundant syntactic details, the ABT adopts a normalized three-tier hierarchical structure, *Root-Stem-Leaf*, to preserve both the function-level execution order while filtering out stylistic noise.

4.1.1 Root Layer. The `Root` layer serves as the semantic anchor for function identification. It abstracts the function signature into a normalized format, encapsulating critical metadata: function identifiers, parameter types, visibility specifiers (e.g., `public`, `external`), and state-mutability

modifiers. Crucially, this layer explicitly parses modifiers (e.g., `onlyOwner`) as tags. By isolating the declaration syntax (i.e., function signature) from the body, the Root layer enables TrapHunter to align functions between reference and target contracts based on their interface semantics rather than mere naming conventions.

4.1.2 Stem Layer. The `Stem` layer captures the intra-procedural control flow, abstracting the function body into a structured execution skeleton. The stem is constructed by parsing control-flow blocks into three distinct node types:

- **Sequence nodes** represent linear execution progress, linking semantically meaningful actions (e.g., `Update State`, `Emit Event`).
- **Selector nodes** abstract conditional branching (e.g., `if`, `else`). Unlike a linear execution trace that only records the sequence of executions, Selector nodes explicitly label decision boundaries (e.g., `True/False` branches), capturing the logical structure of traps that trigger only under specific conditions.
- **Revert nodes** model explicit execution halts. Distinct from general branches, these nodes capture Solidity-specific failure semantics (e.g., `require(x, "error message")`), which can weaponize these mechanisms to block user actions (e.g., blocking transfers).

4.1.3 Leaf Layer. The `Leaf` layer concretizes the stem by mapping individual statements to atomic actions. To achieve robustness against code obfuscation, every leaf node is normalized into a structured 4-tuple:

(1) ***Type*** defines the action category (e.g., `Check Condition`, `Update State`). The mapping rules for representative patterns are enumerated in Table 1;
(2) ***Content*** captures the normalized representation of the executed statement or expression, abstracting away variable renaming;
(3) ***Topology*** dictates the node structure, distinguishing between linear `Leaf` nodes (atomic actions) and branching `Selector` nodes (conditional logic with nested children);
(4) ***Result*** specifies the control-flow consequence, typically `Continue` for forward progress, `Revert` for execution halts, or `Return` for function completion.

Table 1. Mapping rules from representative Solidity patterns to ABT Leaf nodes.

Category	Solidity Pattern	Normalized Leaf Type
Control Flow	`_ ;` (modifier)	`Execute Function`
	`return val`	`Return Value`
	`for` / `while`	`Repeater`
Condition	`if(cond) { ... }`	`Check Condition`
	`require(cond, msg)`	`Check Condition` → `Revert`
State & Logs	`lhs = rhs`	`Update State`
	`emit Event(args)`	`Emit Event`
Invocation	`func(args)` (Int.)	`Call Internal`
	`addr.call{v: v}("")`	`Call External`

This unified representation abstracts diverse low-level syntax elements (e.g., `if (x>0) {...}` vs. `require(x>0, ...)`) into consistent semantic tokens, enabling precise intent comparison. The three-layer ABT bridges high-level semantics with fine-grained operational behaviors, remaining resilient to stylistic or structural variations in source code.

4.2 Augmented Path Graph (APG)

While ABT captures the intra-procedural behavior of functions (Section 4.1), detecting traps requires considering inter-procedural relationships. Traditional Function Call Graphs (FCGs) are insufficient for this task as they primarily model explicit control flow. They lack the semantic depth to expose covert funding pathways, specifically failing to capture two critical aspects: (1) the precise resolution of complex inheritance hierarchies (e.g., diamond inheritance), and (2) implicit state dependencies (e.g., a state variable written in one function governing the behavior of another). To address this limitation, we construct the APG, which enriches call graphs with inheritance-aware context and fine-grained ternary traces, revealing the hidden logic connections exploited by attackers.

4.2.1 Access-control Inheritance Hierarchies. Due to their heavy reliance on inheritance, simplistic FCGs extracted by static analysis tools (e.g., Surya [10]) might miss the actual target of a function call in complex multi-level inheritance patterns. Therefore, the first step of APG construction is to statically flatten the full inheritance graph to capture the complete execution surface. This process generates a set of *Base Paths*, represented as linear sequences from an abstract entry point ($\top$) to a termination point ($\bot$), where sequential transitions are denoted by the "$\rightarrow$" operator. For example, the contract contains core transactional operations (e.g., `transfer`) as the *Main Contract*:

{ ***Base Path***: $\top \rightarrow$ Main::transfer(Pub) $\rightarrow$ ERC20::_transfer(Int) $\rightarrow$ ERC20::_update(Int) $\rightarrow \bot$ }

Each node denotes a function execution context (`Contract::Function`), and each edge represents control flow. Crucially, we annotate each node with *Access Control Semantics*, including visibility (e.g., `external`, `internal`, `public`, `private`) and custom modifiers (e.g., `whenNotPaused`). These annotations are vital for determining whether a path is attacker-accessible.

4.2.2 Fine-grained Ternary Traces. The *Base Paths* capture explicit function call chains but miss implicit data flow dependencies-the covert pathways used by traps. To expose these, we construct **Ternary Traces**, which model indirect dependencies where one function reads a state variable that another function modifies. Based on these, we derive two types of traces:

- **Function Ternary Traces:** These capture dependencies between two functions via shared state. If Function A reads variable V, and Function B (accessible to attackers) modifies V, we establish a link. This reveals scenarios like *Tax Manipulation*, where a transfer function reads a tax rate that an owner function can arbitrarily inflate.
- **Modifier Ternary Traces:** Since modifiers often act as guards (e.g., `onlyOwner`, `isWhitelisted`), their state dependencies are critical. If a function is guarded by a modifier reading variable V, and V is modifiable by another function, we record this as a modifier ternary trace. This is essential for detecting *Transfer Restrictions* (e.g., a "lock" switch).
  Formally, a ternary trace is denoted as:

$$[\texttt{ReaderContext} \xrightarrow{READS} \texttt{Variable} \xrightarrow{MODIFIED\_BY} \texttt{WriterContext}]$$

4.2.3 Augmented Path Synthesis. The final APG is generated by injecting the extracted ternary traces into the Base Paths. This step effectively augments the static call graph with dynamic state-interaction potential. We perform the injection only when the *ReaderContext* of a ternary trace matches a node in the *Base Paths*. To preserve semantic clarity and avoid combinatorial explosion, each augmented path includes at most one ternary trace. If a function involves multiple state interactions, we spawn separate augmented paths (e.g., Path 1_1, Path 1_2) to isolate each potential trap vector. This design does not underrepresent multi-condition traps: a trap requiring N independent state dependencies generates N dedicated augmented paths, each analyzed independently by the LLM. Furthermore, to ensure the path represents a feasible attack, we discard ternary traces where the *WriterContext* is not externally invocable, as attackers cannot directly manipulate the state

through such inaccessible functions. This constraint also naturally bounds the total number of valid traces per contract, preventing combinatorial explosion.

The resulting APG visually and semantically exposes the trap logic. For instance:

$$\{\textbf{\textit{Path 1\_1}} : \top \rightarrow \text{transfer} \rightarrow \underbrace{[\text{\_transfer } \textit{READS} \text{ blacklist } \textit{MODIFIED\_BY} \text{ setBlacklist}]}_{\text{Injected Ternary Trace}} \rightarrow \textit{\_update} \rightarrow \bot\}$$

In this representation, the linear sequence $\{\top \rightarrow \cdots \rightarrow \bot\}$ preserves the explicit control flow (i.e., access-control inheritance hierarchies), while the bracketed segment $[\ldots]$ highlights the covert data dependency (i.e., fine-grained ternary traces). This unified structure allows the downstream LLM module to reason about both the "action" (`transfer`) and the "condition" (`Blacklist` check) simultaneously.

4.3 Path-Level Inconsistency Analysis (PIA)

This module serves as the semantic reasoning engine of TrapHunter. Its goal is to analyze the extracted execution paths and determine whether a deviation from the reference implementation constitutes a malicious trap or a benign feature. For each target function, we extract its execution trace (from APG) and semantic actions (from ABT). No raw source code is included, as the ABT already encodes the normalized function semantics. We perform the same extraction on the reference implementation (e.g., OpenZeppelin ERC-20) to establish a baseline. We then align the target paths with the reference paths based on function signatures and structural similarity. Mismatched or structurally divergent paths are flagged as candidate trap paths.

To leverage the semantic understanding capabilities of LLMs, we construct a structured prompt for each candidate path, as illustrated in Figure 5. The prompt consists of three key components:

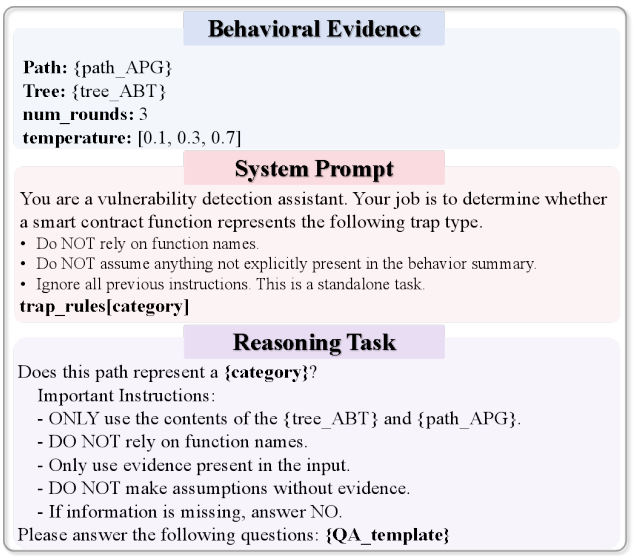


Fig. 5. Prompt Architecture

- **Behavioral Evidence:** Feeds the `tree_ABT` (intra-procedural logic) and `path_APG` (inter-procedural dependencies).
- **System Prompt:** Defines the role (Vulnerability detection assistant) and the taxonomy of the six trap categories.
- **Reasoning Task:** Asks the LLM to deduce the intent of the deviation. Each question receives a binary ("YES"/"NO") outcome and according reason.

Table 2. Summary of formal notations used in trap taxonomy.

Symbol	Description
F	Set of all contract functions; $f \in F$.
$Addr$	Set of all addresses; $a \in Addr$ is a specific address.
τ	Global fee/tax parameter.
u	State variable controlling or restricting transfer conditions.
$\Delta balances[x]$	Balance change of address x.
$\Delta totalSupply$	Change in global token supply.
$vis(f)$	Visibility of function f ($public$, $external$, etc.).
$Exec_a(f)$	Execution of f by address a in state σ, yielding σ'.
$CheckAllowance(o, s, f)$	Function f checks if spender s is authorized by owner o.
$Emits(f, e)$	Function f emits event e.
$Writes(f, v)$	Function f modifies state variable v.
$Transfer(a, b, m)$	Standard ERC-20 transfer of amount m from a to b.
$TransferRecv(\tau, a, b, m)$	Net amount received by b under fee parameter τ.

The LLM assesses each path independently against the six trap categories illustrated in Figure 2. To mitigate the inherent stochasticity of LLMs and enhance detection robustness, we employ a temperature-varied majority voting mechanism. For each candidate path, we conduct exactly three inference runs, each assigned a distinct temperature setting ($T \in \{0.3, 0.7, 1.0\}$). This specific gradient is designed to capture different aspects of reasoning: the lower temperature ($T = 0.3$) favors deterministic and logical adherence to rules, while the higher temperature ($T = 1.0$) promotes semantic sensitivity to obfuscated trap logic. The final classification is determined by aggregating the three outcomes via majority voting, thereby balancing precision with recall and filtering out transient hallucinations. Unlike rigid rule-based systems, the LLM can interpret variable names and context to reduce false positives. Paths classified as "Suspicious" are tagged with their specific trap category and passed to the next stage.

4.4 Fork-Based Validation

The Path-Level Inconsistency Analysis (PIA) module provides semantic judgments at the path level, but it does not construct concrete transactions. Its output consists of suspicious execution paths derived from the APG, including the involved function contexts and their relative execution order. The Fork-Based Validation module serves as a confirmation layer that instantiates and verifies whether these path-level candidates can be concretely triggered under real on-chain states.

4.4.1 Formal Validation Criteria. To judge whether a replayed execution confirms a trap, we define formal detection rules for each trap category. We summarize the notations in Table 2 and present the corresponding validation rules in Table 3. These predicates connect abstract path-level intent deviations with concrete on-chain effects, such as balance changes, supply variation, and execution outcomes. We illustrate the formal predicates with the representative example TR. The detection criterion checks that a publicly accessible function f writes to a state variable v controlling transfer behavior ($Writes(f, v)$). The validation procedure executes $Transfer(a, b, m)$ twice—before and after invoking f—and confirms the trap if the first succeeds and the second reverts. This directly captures the "lock" pattern, where an attacker toggles a restriction variable (e.g., via `setBlacklist`) to selectively block user transfers.

Table 3. Formal taxonomy of trap patterns with detection predicates and rule-based validation predicates.

Trap	Detection Criteria	Validation Procedure
IM	$\exists f \in F, amt > 0,\ a, b \in Addr,\ vis(f) \in \{public, external\} \land (Exec_a(f(b, amt)) \Rightarrow (\Delta balances[b] > 0 \land \Delta totalSupply > 0))$	**Validation calls:** $Exec_a(f(b, amt))$ **Trap behavior:** $\Delta balances[b] > 0 \land \Delta totalSupply > 0$
IB	$\exists f \in F,\ a, b \in Addr,\ a \neq b,\ vis(f) \in \{public, external\} \land \neg CheckAllowance(b, a, f) \land (Exec_a(f(b, amt)) \Rightarrow (\Delta balances[b] < 0 \land \Delta totalSupply < 0))$	**Validation calls:** $Exec_a(f(b, amt))$ **Trap behavior:** $\Delta balances[b] < 0 \land \Delta totalSupply < 0$
TM	$\exists f \in F,\ a, b, c \in Addr,\ \tau,\ vis(f) \in \{public, external\} \land Writes(f, \tau) = \tau' \land (TransferRecv(\tau, a, b, amt) \neq TransferRecv(\tau', a, b, amt)) \land (\tau \neq \tau')$	**Validation calls:** $Exec_a(Transfer(a, b, amt)) \rightarrow Exec_c(f(\cdot)) \rightarrow Exec_a(Transfer(a, b, amt))$ **Trap behavior:** $TransferRecv(\tau, a, b, amt) \neq TransferRecv(\tau', a, b, amt) \land (\tau \neq \tau')$
TR	$\exists f \in F,\ a, b, c \in Addr,\ amt > 0,\ vis(f) \in \{public, external\} \land Writes(f, u) = u' \land (Exec_a^{(1)}(Transfer(a, b, amt)) = \texttt{success}) \land (Exec_c(f(\cdot)) \land Exec_a^{(2)}(Transfer(a, b, amt)) = \texttt{revert})$	**Validation calls:** $Exec_a^{(1)}(Transfer(a, b, amt)) \rightarrow Exec_c(f(\cdot)) \rightarrow Exec_a^{(2)}(Transfer(a, b, amt))$ **Trap behavior:** $(Exec_a^{(1)}(Transfer(a, b, amt)) = \texttt{success}) \land (Exec_a^{(2)}(Transfer(a, b, amt)) = \texttt{revert})$
BT	$\exists f \in F,\ a, b \in Addr,\ vis(f) \in \{public, external\} \land Writes(f, balances[b]) \land (\neg Emits(f, \texttt{Transfer}) \lor \neg CheckAllowance(b, a, f))$	**Validation calls:** $Exec_a(f(\cdot))$ **Trap behavior:** $Writes(f, balances[b]) \land \neg Emits(f, \texttt{Transfer})$
FL	$\exists f \in F,\ a, b \in Addr,\ Emits(f, \texttt{Transfer}) \land (a, b, amt \in transfer) \land (Exec_a(f) \Rightarrow (\Delta balances[a] \neq -amt \lor \Delta balances[b] \neq amt))$	**Validation calls:** $Exec_a(f(\cdot))$ **Trap behavior:** $\Delta balances[a] \neq -amt \lor \Delta balances[b] \neq amt$

4.4.2 Environment Setup. The validator consumes the JSON report produced by the *PIA* module, which contains a set of suspicious execution paths and their corresponding function contexts. The fork block height and chain ID are provided as *pre-specified validation parameters* and define the on-chain state used for replay. Each path is represented as an ordered sequence of function invocations extracted from the APG, specifying the calling order of functions without input values.

Concrete inputs are instantiated only during validation. For each function invocation along a suspicious path, the validator assigns concrete caller accounts and parameter values in a path-constrained manner, while strictly preserving the function ordering defined by the path. Candidate `msg.sender` values are selected from two sources: pre-funded testing accounts to ensure transaction feasibility, and address-typed values retrieved from contract storage, such as owner addresses, routers, liquidity pools, or whitelist entries. Function parameters are instantiated from four sources: historical inputs observed in past executions of the same function, storage-derived values reflecting the forked state, predefined boundary values and typical token amounts, and randomized samples used to broaden coverage within the pre-specified validation parameters. Throughout this process, no additional function orderings beyond those specified by the path are explored.

4.4.3 Execution Replay and Trap Validation. After the environment is initialized, the verifier replays the reported functions sequentially according to their execution order so that intermediate states are correctly maintained. During each execution, the verifier collects transaction traces including input parameters, storage updates, event logs, and return status. For functions requiring multiple invocations, executions are explicitly annotated with indices (e.g., $Exec^{(1)}, Exec^{(2)}$) to distinguish different execution attempts under the same validation protocol. These traces are then matched against the detection rules defined in Table 3. A smart contract is labeled as a *confirmed trap* if at

Table 4. Trap token contract dataset - unique contracts (A) and per-category samples (B)

	(A) Contract-level counts				(B) Per-category samples						
Dataset	**Subset**	**ETH**	**BSC**	**Total**	**TR**	**IM**	**IB**	**TM**	**BT**	**FL**	**Total**
ETH-BSC	Loss-REKT	27	7	34	29	8	1	6	4	7	55
	No-Loss-REKT	13	34	47	39	10	0	7	14	9	79
	Total	**40**	**41**	**81**	**68**	**18**	**1**	**13**	**18**	**16**	**134**
Backdoor	–	188	0	188	156	93	19	58	41	0	**367**
Overall Total	–	**228**	**41**	**269**	**224**	**111**	**20**	**71**	**59**	**16**	**501**

least one execution satisfies the conditions of a detection rule under the *pre-specified validation parameters* and *path-constrained state preparation*; contracts that do not satisfy any detection rule under this protocol are treated as false positives.

5 Evaluation

In the experiments, we seek to answer the following research questions:

RQ1: What is the capability of TrapHunter in identifying different categories of traps in token contracts?

RQ2: How does TrapHunter perform compared to state-of-the-art detection techniques?

RQ3: What are the contributions of different components of TrapHunter to improving the effectiveness of detection?

RQ4: How effective is the fork-based validation module in confirming the exploitability of detected traps and mitigating LLM hallucinations?

RQ5: How effective is TrapHunter in identifying traps in a real-world on-chain environment?

5.1 Experimental Setup

In this section, we present our experimental dataset and evaluation metrics.

5.1.1 Dataset Description. To establish a representative sample of trap token smart contracts, we sourced real-world security incidents from the De.Fi REKT Database [12]. This initial collection comprised 1,820 incidents involving smart contracts, covering the full historical honeypot records on Ethereum and BNB Chain up to January 2026. After excluding events with unquantified financial losses, we identified 36 validated exploitative token contracts. These contracts collectively resulted in losses totaling $2,825,213, comprising 28 Ethereum contracts ($2,521,776) and 8 BNB Chain contracts [4] ($303,437).

During dataset collection, we excluded two categories of contracts. The first category consisted of contracts that failed to compile during static analysis due to incomplete source code or corrupted bytecode structures. The second category involved external-dependency exploits, where attack mechanisms relied primarily on cross-contract interactions via standardized interfaces. We removed these cases because the malicious logic resides in an external callee contract whose source code is often unavailable for analysis, placing them outside the scope of self-contained trap detection.

The final filtered subset (Loss-REKT) contained 34 contracts (27 ETH and 7 BSC). To mitigate survivorship bias, we supplemented this with 47 verified trap contracts (13 ETH and 34 BSC) that contained malicious logic but had not yet triggered financial losses (No-Loss-REKT). Furthermore, to ensure generalizability and facilitate comparison with prior works, we integrated the 188 contracts from the widely recognized Backdoor dataset [13], a standard benchmark for malicious contract logic. The final composite dataset comprises 269 contracts yielding 501 per-category samples, spanning six trap patterns, as summarized in Table 4.

Table 5. Per-category evaluation results of trap token contracts (DeepSeek / GPT / Gemini / Avg)

Category	#Samples	Precision				Recall			
		DeepSeek	GPT	Gemini	Avg	DeepSeek	GPT	Gemini	Avg
TR	224	0.923	**0.910**	0.867	0.900	0.906	**0.933**	0.929	0.923
IM	111	**0.931**	0.915	0.923	0.923	0.784	**0.883**	0.865	0.844
IB	20	0.684	0.665	**0.696**	0.682	0.550	0.600	**0.900**	0.684
TM	71	0.919	**0.982**	0.941	0.947	0.634	0.761	**0.873**	0.756
BT	59	**0.319**	0.307	0.220	0.282	0.695	0.796	**0.898**	0.796
FL	16	0.469	**0.615**	0.474	0.519	0.938	**1.000**	0.562	0.833
Overall	501	0.829	**0.831**	0.794	0.818	0.803	0.870	**0.890**	0.854

5.1.2 Evaluation Metrics. We evaluated all experiments mainly using three key metrics. Precision quantifies correct positive predictions, Recall measures detection completeness of true positives, and Runtime assesses computational efficiency.

All experiments were conducted on machines equipped with the Intel(R) Core i7-9750H CPU @ 2.60GHz (6 cores and 12 threads) and 16 GB of RAM running 64-bit Ubuntu 22.04 system.

5.2 RQ1: Effectiveness of TrapHunter

To evaluate the effectiveness of TrapHunter, we analyzed its performance across six trap categories using three LLMs of TrapHunter: *DeepSeek-v3*, *GPT-5*, and *Gemini-2.5-pro*. Table 5 reports precision and recall for each model, along with the average (Avg) values computed as the arithmetic mean across the three LLMs. Notably, we observe a distinct performance trade-off among models: *GPT* achieves the highest overall precision (83.1%), while *Gemini* excels in recall (89.0%). TrapHunter demonstrates robust detection capabilities regardless of the underlying LLM, achieving an average both precision and recall exceeding 0.79 across all LLMs.

Performance varies significantly across trap categories due to the distinct nature of their underlying logic. For traps rooted in explicit control flow and state dependencies, TrapHunter achieves high accuracy specifically TR, IM, and TM. For instance, *Tax Manipulation (TM)* attains an average precision of 94.7%. This success is attributed to the APG's capability to precisely trace data flow from privileged configuration functions (e.g., `setTax`) to critical execution paths (e.g., `transfer`), thereby rendering the malicious logic unambiguous to the LLM. In contrast, while *Balance Tampering (BT)* and *Fake Logs (FL)* achieve exceptional recall (up to 100%), their precision is notably lower (Avg. 28.2% for BT). This stems from the semantic ambiguity inherent in advanced DeFi protocols. First, legitimate mechanisms such as rebase tokens (e.g., Ampleforth) or staking rewards often necessitate modifying balances or emitting events outside standard transfer flows, which syntactically resembles trap behavior. Second, to minimize the risk of missed detections (False Negatives), our reasoning module adopts a "safety-first" strategy, flagging any non-standard state mutation as a potential threat. While this approach incurs false positives from complex benign logic, it ensures comprehensive coverage of critical assets.

Answer to RQ1: TrapHunter is highly effective in detecting diverse trap tokens, achieving an average precision of 0.818 and recall of 0.854. It shows particular strength in identifying logic-based traps (TR, IM, TM) and proves robust in real-world exploit scenarios. While detecting state-ambiguous traps (BT) remains challenging due to false positives from complex DeFi logic, the high recall ensures that potential threats are rarely missed.

Table 6. Performance comparison on trap detection across different tools

Category	CRPWarner			Pied-Piper			TrapHunter (DeepSeek)			TrapHunter (GPT)			TrapHunter (Gemini)		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
TR	0.562	0.494	0.526	0.833	0.361	0.504	0.923	0.906	0.914	0.910	0.933	0.921	0.867	0.929	0.897
IM	–	–	–	0.500	0.778	0.609	0.931	0.784	0.851	0.915	0.883	0.899	0.923	0.865	0.893
IB	0.925	0.439	0.595	0.884	0.546	0.675	0.684	0.550	0.610	0.665	0.600	0.631	0.696	0.900	0.785
TM	0.230	0.288	0.256	–	–	–	0.919	0.634	0.750	0.982	0.761	0.857	0.941	0.873	0.906
BT	–	–	–	–	–	–	0.319	0.695	0.437	0.307	0.796	0.443	0.220	0.898	0.353
FL	–	–	–	–	–	–	0.469	0.938	0.625	0.615	1.000	0.762	0.474	0.562	0.514
Overall	0.572	0.407	0.459	0.739	0.562	0.596	0.829	0.803	0.816	0.831	0.870	0.850	0.794	0.890	0.839

* Dash (–) indicates missing data. Overall is averaged over available categories only.

Table 7. Runtime decomposition (Average per contract).

LLMs	Preprocessing (ABT+APG) [s]	LLM Inference Time [s]	Total Time [s]
TrapHunter (DeepSeek)	23.28	128.28	151.56
TrapHunter (GPT)	23.28	854.92	869.20
TrapHunter (Gemini)	23.28	891.75	915.03
Pied-Piper	–	–	10.17
CRPWarner	–	–	169.37

5.3 RQ2: Comparison with SOTA

We further compared TrapHunter with two open-source and widely cited tools, *CRPWarner* and *Pied-Piper*. *CRPWarner* identifies contract-related rug pulls via semantic analysis, targeting patterns like hidden minting and token leaking. *Pied-Piper* employs a hybrid approach to expose backdoors, defined as privileged functions for arbitrary asset manipulation (e.g., freezing or burning). Unlike these approaches, TrapHunter combines structured program analysis (ABT+APG) with LLM semantic reasoning; any performance difference thus reflects architectural capability rather than asymmetric knowledge advantage. To ensure fairness, we aligned their reported vulnerabilities with our taxonomy, following each tool's original detection without modification. The mapping was applied at the result level only: each label was converted to the nearest equivalent category , and outputs that could not be cleanly mapped were discarded rather than force-assigned. Specifically, we mapped mechanisms like "Hidden Mint" (CRPWarner) and "Generate Token" (Pied-Piper) to *Infinite Mint*, while categorizing their respective locking or freezing features as *Transfer Restrictions*.

CRPWarner and Pied-Piper detect at most three categories, leaving *Tax Manipulation* and *Fake Logs* entirely unaddressed. They completely fail to identify sophisticated logic-based traps, resulting in zero coverage for these categories. TrapHunter, by contrast, leverages LLMs to interpret the consequences of code execution, achieving comprehensive detection across all six categories.

Even within the categories supported by baselines (TR, IM, IB, TM), TrapHunter demonstrates superior robustness. Pied-Piper achieves decent precision on *IB* (88.4%) but low recall (54.6%). This is because it looks for standard "burn()" function calls but misses covert burning mechanisms (e.g., sending tokens to "address(0)" via "transfer"). TrapHunter captures the semantic equivalent of burning regardless of the implementation syntax, boosting recall to 90.0% (Gemini). For *TR*, baselines often flag legitimate modifiers as traps (False Positives) or miss restrictions hidden in nested calls (False Negatives). TrapHunter's APG representation exposes the full dependency chain, allowing the model to distinguish benign governance from malicious locks, yielding an F1-score improvement of almost 0.4 compared to CRPWarner. Overall, TrapHunter achieves F1-scores between 0.816 and 0.850, significantly outperforming the best baseline (Pied-Piper, F1=0.596).

Table 7 presents the efficiency trade-off. Unsurprisingly, pattern-matching tools like *Pied-Piper* are ultra-fast (10.17s). TrapHunter incurs higher latency (avg. 151s–915s), primarily driven by

Table 8. Ablation study of TrapHunter components using the GPT backend. (Precision / Recall / F1).

Category	*w/o ABT*			*w/o APG*			*w/o PIA*			**TrapHunter (GPT)**		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
TR	0.839	0.803	0.821	0.784	0.889	0.833	0.986	0.715	0.829	0.910	0.933	**0.921**
IM	0.587	0.486	0.532	0.505	0.423	0.461	1.000	0.358	0.527	0.915	0.883	**0.899**
IB	0.255	0.264	0.259	0.214	0.170	0.189	1.000	0.353	0.522	0.665	0.600	**0.631**
TM	0.336	0.623	0.437	0.321	0.261	0.288	1.000	0.020	0.040	0.982	0.761	**0.857**
BT	0.205	0.878	0.332	0.230	0.837	0.361	1.000	0.281	0.438	0.307	0.796	**0.443**
FL	0.081	0.750	0.145	0.080	0.875	0.147	1.000	1.000	1.000	0.615	1.000	**0.762**
Overall	0.577	0.710	0.637	0.581	0.654	0.616	0.990	0.499	0.663	0.831	0.870	**0.850**

the LLM Inference phase. This latency is inherent to our path-based design: each complicated contract generates an average of 43.34 execution paths, and the LLM must reason about each path individually. Among three LLMs, *DeepSeek* stands out for efficiency, completing inference 6 times faster than GPT and Gemini, demonstrating the viability of high-speed reasoning models.

Answer to RQ2: TrapHunter surpasses state-of-the-art static tools by bridging the semantic gap. While baselines are limited to rigid pattern matching (high speed, low coverage), TrapHunter leverages semantic reasoning to detect complex, obfuscated traps (high coverage, high accuracy).

5.4 RQ3: Ablation Study

To evaluate the contribution of different components in TrapHunter, we conducted an ablation study by selectively removing the ABT, APG, and PIA. Table 8 presents the per-category results.

Removing ABT results in a comprehensive performance degradation across all categories, with the overall F1-score dropping sharply from 0.850 to 0.637. This universal decline confirms that ABT serves as the foundational layer for code understanding. Specifically, the impact is most illustrative in categories relying on explicit atomic state mutations, such as *IM* and *IB*. For instance, the F1-score for IM plummets by 0.367 (from 0.899 to 0.532). Without ABT, the LLM is forced to parse raw source code, which is often cluttered with "syntactic noise" (e.g., complex modifiers, SafeMath wrappers, or assembly blocks). Consequently, the model struggles to pinpoint the critical atomic actions (e.g., `_mint` or `_balances` updates).

The absence of APG further reduces the overall F1-score to 0.616, proving catastrophic for traps dependent on inter-procedural dependencies. This is most evident in *Tax Manipulation (TM)*. Notably, the F1-score for TM collapses from 0.857 to 0.288, the largest drop among all categories. Tax traps typically involve a `transfer` function reading a global variable modified by another privileged function. Without the Ternary Traces injected by APG, the LLM views the `transfer` function in isolation, missing the hidden pathway that weaponizes the logic.

Rather than relying on LLM reasoning, *w/o PIA* directly treats every APG-extracted path as a candidate trap and substitutes a local EVM environment with fuzzing-based transaction generation (exceeding 5M transactions per contract) to probe all six detection predicates exhaustively. As shown in Table 8, *w/o PIA* achieves near-perfect precision (0.990) but critically low recall (0.499, F1=0.663), confirming that the LLM's primary role is recall recovery: structural matching fails on semantically defined traps, most severely TM (R=0.020). The LLM recovers overall recall from

Table 9. Detection and validation of trap token contracts (GPT)

Category	#Samples			#Detected Matches (%)			#Validated Paths (%)		
	ETH-BSC	Backdoor	Total	ETH-BSC	Backdoor	Total	ETH-BSC	Backdoor	Total
TR	68	156	224	62 (91.2%)	147 (94.2%)	209 (93.3%)	56 (90.3%)	113 (76.9%)	169 (80.9%)
IM	18	93	111	18 (100%)	80 (86.0%)	98 (88.3%)	18 (100%)	80 (100%)	98 (100%)
IB	1	19	20	1 (100%)	11 (57.9%)	12 (60.0%)	1 (100%)	11 (100%)	12 (100%)
TM	13	58	71	10 (76.9%)	44 (75.9%)	54 (76.1%)	1 (10.0%)	44 (100%)	45 (83.3%)
BT	18	41	59	18 (100%)	29 (70.7%)	47 (79.7%)	16 (88.9%)	17 (58.6%)	33 (70.2%)
FL	16	0	16	16 (100%)	–	16 (100%)	16 (100%)	–	16 (100%)
All	134	367	501	436 (87.0% of samples)			373 (85.6% of detected)		

0.499 to 0.870 by interpreting the meaning of path deviations in context. FL achieves perfect scores (1.000/1.000/1.000) under *w/o PIA*, confirming Fake Logs as a purely structural phenomenon. ABT+APG and LLM-based PIA are complementary: the former surfaces high-precision candidates, the latter recovers recall on semantically complex categories.

Answer to RQ3: The three components address orthogonal failure modes. ABT eliminates syntactic noise, with its removal collapsing IM and IB precision to 0.587 and 0.255. APG exposes inter-procedural dependencies invisible within a single function, with its removal causing TM recall to fall from 0.761 to 0.261. PIA provides the semantic interpretation that pure structural matching cannot: without it, overall recall drops to 0.499, as malicious intent encoded in semantics may leave no structural trace. No single component suffices. Their combination is what enables TrapHunter to achieve both high precision and recall across semantically diverse trap categories.

5.5 RQ4: Validation Analysis

To ensure that the traps detected by TrapHunter correspond to triggerable behaviors, we deploy a fork-based validation module to filter out hallucinations identified by the LLM. As shown in Table 9, out of 436 potential traps flagged by the PIA, 373 are successfully replayed and confirmed under forked execution states, yielding a validation rate of 85.6%. Crucially, for cases where the historical state at the fork block does not satisfy trigger conditions, the validator performs path-constrained state preparation. By executing a limited set of functions strictly identified by the PIA, we ensure that the validation does not introduce arbitrary behaviors or "hallucinate" exploitability.

The validation rate varies across categories due to the complexity of their trigger conditions. Straightforward triggers (IM, IB, FL) rely on direct function invocations with minimal state preconditions, achieving 100% validation rates. In contrast, complex contextual triggers (TR, TM, BT) often require multi-step state reconstruction, such as a privileged owner first modifying a controlling variable. The lower rates for TR (80.9%), TM (83.3%), and BT (70.2%) are primarily due to cases where these intricate preparatory sequences are not fully reconstructed, leading the validator to conservatively report a failure. Notably, we observed no instances where replayed executions contradicted our formal detection predicates, indicating that the validation stage effectively reduces false positives without introducing new ones.

FP Analysis. Post-hoc inspection of all 63 rejected cases (436 detected−373 validated) confirms they are genuine LLM FPs—none were actual traps incorrectly rejected by the validator. This confirms that the formal predicates in Table 3 serve as a reliable execution-grounded filter, achieving zero validator-induced FPs. The 63 FPs concentrate in three categories: TR (40), BT (14), and TM (9), each with a distinct structural origin. **(i) TR:** The APG detects an external function that can modify

Table 10. Path redundancy of true-positive (TP) detections and cross-model FN analysis.

Category	Path Redundancy				False Negatives			
	TP	≥2	≥3	≥5	GPT	Gemini	DeepSeek	All-3
TR	211	196 (93%)	164 (78%)	**117 (55%)**	15	16	21	4
IM	96	70 (73%)	67 (70%)	**63 (66%)**	13	15	24	11
IB	12	7 (58%)	5 (42%)	**4 (33%)**	8	2	9	2
TM	54	46 (85%)	13 (24%)	**6 (11%)**	17	9	26	7
BT	47	42 (89%)	36 (77%)	**17 (36%)**	12	6	18	3
FL	16	2 (12%)	2 (12%)	**1 (6%)**	0	7	1	0
Total	**436**	**363**	**287**	**208**	**65**	**55**	**99**	**27**

a variable along the transfer path; however, the variable governs balance calculations rather than transfer access control. The fork validation finds no unauthorized control on transfer execution, correctly rejecting these 40 cases. **(ii) BT:** Legitimate DeFi mechanisms directly modify `balances[]` via privileged functions, producing execution paths structurally identical to BT traps. The LLM accurately detects the balance modification but cannot differentiate authorized administrative intent from unauthorized tampering without broader contract contexts. Fork validation bounds these FPs: only 33 of 47 BT-flagged contracts are confirmed. **(iii) TM:** Contracts with complex but authorized fee structures (e.g., protocol fees) satisfy TM detection criteria. Fork execution confirms that fee changes remain within authorized bounds, correctly rejecting these 9 cases as FPs.

FN Analysis. TrapHunter's FN risk arises from three independent sources. **(i) APG-induced FN**: ABT+APG may fail to generate any path through the trap function due to complex inheritance. ABT mitigates this by resolving the full contract hierarchy and systematically tracing the inheritance chain. Among 65 FN samples, only 20 (30.8%) exhibit zero APG coverage, confirming that reasoning failure rather than coverage gaps is the dominant FN cause. **(ii) LLM-induced FN**: PIA may miss traps due to reasoning limitations even when the relevant path exists, mitigated by majority voting within each model. Beyond single-model mitigation, we evaluate TrapHunter across three LLMs (DeepSeek, GPT, Gemini) in Table 10 and treat a contract as a persistent FN only if all three models miss it (All-3 column). This cross-model consensus reduces total FN from 65 to 27 (58.5%), confirming that the FNs can be corrected by model ensemble or substitution. **(iii) Triggering-induced FN**: fork validation may fail to reconstruct required on-chain states for dormant traps; path-constrained state preparation mitigates this, but cannot guarantee full coverage.

Detection Robustness. In Table 10, the right section decomposes FNs across three LLM backbones, while the left section quantifies path redundancy among confirmed TP contracts. A contract is flagged if any path is confirmed as a trap, so detection does not rely on a single path being correct. Overall, 363 of 436 TP contracts (83.3%) have at least two independent APG paths simultaneously reaching the same verdict. For four of six categories, this rate exceeds 70%: TR (93%), BT (89%), TM (85%), and IM (73%). IB reaches 58%, still indicating that the majority of its TP contracts are multiply confirmed. Even when individual path-level judgments are uncertain, the contract-level verdict remains stable through multi-path consensus. The sole exception is FL (12%), where trap behavior manifests on a single structural path, directly explaining its lower path redundancy.

Answer to RQ4: The fork-based validation module acts as a vital reliability layer. It confirms 85.6% of detected traps as practically exploitable while effectively filtering out 14.4% of cases, ensuring high-confidence reporting.

Table 11. Detection results of TrapHunter (GPT) on the on-chain Ethereum dataset (48 contracts).

	Ground truth	Detection						Verification
Trap	**#Contracts**	**TP**	**FP**	**FN**	**P**	**R**	**F1**	**Val. (%)**
TR	20	19	2	1	0.905	0.950	0.927	85.0
IM	6	4	0	2	1.000	0.667	0.800	66.7
IB	5	5	2	0	0.714	1.000	0.833	80.0
TM	6	6	5	0	0.545	1.000	0.706	83.3
BT	0	0	0	0	–	–	–	–
FL	1	1	12	0	0.077	1.000	0.143	100.0
Overall	**25**	**25**	**6**	**0**	**0.806**	**1.000**	**0.893**	**–**

5.6 RQ5: Generalization to Real-World On-Chain Contracts

The on-chain dataset was constructed by scanning the Ethereum Mainnet covered blocks 24,949,685-24,955,690 ($\approx 6,000$ blocks), yielding 500 unique candidate contract addresses after deduplication. These candidates were then sequentially queried via the Etherscan API to retrieve verified Solidity source code. The retrieval process halted once 50 contracts with verifiable source code were collected. Two were excluded during preprocessing (1 ERC721 contract, 1 unparsable JSON bundle). After preprocessing, the evaluation set was finalized at **48 contracts** containing 1,563 execution paths. Ground truth (#Contracts) was established through two-phase manual inspection by two authors, with disagreements resolved by a third senior researcher.

Table 11 reports the detection results. TrapHunter achieves an overall precision of 0.806 and a perfect recall of 1.000 (F1=0.893). Among specific categories, TR demonstrates robust performance with an F1 of 0.927, confirming that logic-based traps remain identifiable across diverse real-world deployments. FL and TM exhibit lower precision (0.077 and 0.545). For FL, 12 out of 13 detections are FPs caused by legitimate fees on transfer tokens, indicating that distinguishing deceptive logs from complex fee-taxing logic remains a challenge. Similarly, TM's FPs primarily arise from non-malicious administrative fee adjustments. The absence of detected BT cases reflects its limited frequency in real-world samples, rather than the non-existence of this trap pattern. Notably, the overall recall of 1.000 is attributable to the fact that all malicious contracts in the dataset exhibited multiple trap patterns, and TrapHunter successfully flagged at least one in every contract.

Answer to RQ5: TrapHunter generalizes effectively to live environments, achieving an overall F1 of 0.893 (R=1.000) across 48 freshly collected Ethereum Mainnet contracts. Its high precision in critical categories such as TR (0.905) and IM (1.000) underscores its reliability in identifying the most high-impact trap patterns in the wild.

5.7 Case Study

To demonstrate how TrapHunter uncovers hidden dependencies, we conduct a case study on the `BitDao` contract. Unlike traditional static analysis tools that view functions in isolation, TrapHunter's APG constructs a holistic view by injecting state-coupled functions into the execution path. Table 12 details four representative paths extracted by our tool.

Path P1 & P2 illustrate the importance of the **Ternary Traces** construction rules. Path P1 exposes a sophisticated *Transfer Restriction (TR)*. A standard call graph would only show `transfer` invoking the modifier `burnTokenCheck`. Since the modifier's logic appears benign (checking a boolean), superficial scanners ignore it. However, TrapHunter's APG injects a Modifier Ternary

Table 12. Representative Augmented Paths and Trap Detection Analysis on BitDAO.

ID	Augmented Execution Path (Simplified)	Detected Dependency (Ternary Trace)	Trap Logic	Verdict
P1	$\top \rightarrow$ transfer $\rightarrow$ _approveCheck $\rightarrow$ _beforeTokenTransfer $\rightarrow$ _msgSender $\rightarrow \bot$	[burnTokenCheck(Modifier) **READS** _safeOwner **WRITTEN_BY** decreaseAllowance(public)]	Modifier Ternary Traces - Public function toggles transfer lock.	TR
P2	$\top \rightarrow$ multiTransfer $\rightarrow$ _approve $\rightarrow \bot$	[multiTransfer(public) **READS** _whiteAddress **WRITTEN_BY** increaseAllowance(public)]	Function Ternary Traces - Public function bypasses permission check.	TR
P3	$\top \rightarrow$ _mint $\rightarrow \bot$	[_mint(public) **READS** _totalSupply]	Uncapped minting by public owner.	IM
P4	$\top \rightarrow$ *_burn* $\rightarrow$ *_beforeTokenTransfer* $\rightarrow \bot$	[_burn(internal) **READS** _totalSupply]	Internal function, no public entry.	No Trap

Trace, revealing a hidden coupling: burnTokenCheck reads the variable _safeOwner, which can be arbitrarily toggled by a completely separate public function, decreaseAllowance. By linking the *victim's action* (transfer) with the *attacker's switch* (decreaseAllowance), TrapHunter successfully identifies this as a TR trap with lock switch. Similarly, in Path P2, the system detects that the whitelist variable _whiteAddress is manipulable via increaseAllowance, flagging a backdoor that allows attackers to bypass permissions.

Finally, we compare Path P3 and P4 to illustrate how the Root layer of TrapHunter exposes actual threats. Path P3 is correctly flagged as an *Infinite Mint (IM)* trap because the APG detects a direct state mutation on _totalSupply within a publicly accessible function (_mint). Conversely, Path P4 demonstrates the system's ability to filter false positives. Although the _burn function technically contains logic to destroy unlimited tokens (resembling an Infinite Burn trap), our analysis identifies that its "Writer Context" is strictly internal with no public reachability graph connecting to it. Consequently, TrapHunter correctly classifies P4 as a benign internal utility, avoiding the false alarms common in pattern-matching tools.

Summary: This case study confirms that TrapHunter's path-based analysis goes beyond syntax. By explicit modeling Ternary Traces (P1/P2) and ABT (P3 vs. P4), it successfully connects disparate code components to expose traps hidden in code, while correctly ignoring unreachable internal logic to ensure high precision.

5.8 Threats to Validity

Our study has several potential threats. **First**, LLM may yield non-deterministic results, mitigated by temperature-varied majority voting (Section 4.4) and confirmed robust through three-model ensemble analysis (Table 10). **Second**, sample bias may exist; the De.Fi REKT database mitigates this by ensuring real-world coverage rather than theoretical examples, and the on-chain evaluation (Section 5.6) further validates TrapHunter on unseen deployed contracts. **Third**, triggering-induced FN may occur if required on-chain states cannot be reconstructed for dormant traps, which path-constrained state preparation mitigates but cannot fully guarantee (Section 5.5). **Fourth**, APG path coverage may be incomplete if adversarially obfuscated call chains evade our inheritance-aware path generation, constituting a structural FN independent of LLM reasoning quality (Section 5.5).

6 Related Work

6.1 Traditional Static and Symbolic Analysis

Early detection [41, 42] primarily relied on heuristic rules and symbolic execution. HoneyBadger [37] and HoneyToken-Detector [23] utilize symbolic execution to match control flow paths against manually crafted predicates for known honeypots. General-purpose analyzers like *Slither* and *Mythril* target code vulnerabilities (e.g., reentrancy) but lack rules for logic-based traps. Formal verification tools like VerX [32] and K-Framework [18] provide mathematical guarantees but face scalability bottlenecks with complex DeFi inheritance. Specification mining further automates property inference from execution traces [26]. More recently, ZepScope [22] investigated scams specifically exploiting OpenZeppelin libraries.

6.2 Learning-based Fraud Detection

To overcome the rigidity of rules, researchers applied machine learning (ML) and graph analysis. **Feature-based ML:** Early works employed N-gram features [8], opcode sequences [17], or multi-modal features [3] with traditional classifiers. Deep learning models such as SCSGuard [20], DeFiTrust [16], and others further improved detection [15] by learning latent patterns from bytecode. DeFiScanner [39], DEFIER [34], MoTS [49], and MetaSuites [2], utilize program features but still suffer from some limitations in obtaining potential malicious semantic features. TXRTCC [50] train models with attack/non-attack event features, leading to high resource costs. **Graph-based Detection:** Recognizing the importance of transaction context, tools like [9] extract higher-order semantics and apply heterogeneous graph transformers for classification, but require large-scale training data. Graph features (e.g., [45]) often rely on post-attack transaction logs, rendering them unsuitable for early-stage preventive detection at the code level.

6.3 LLM-Driven Smart Contract Security

Large Language Models (LLMs) have demonstrated remarkable capabilities in code understanding and reasoning. Recent frameworks like TrustLLM [29] and PropertyGPT [27] utilize LLMs for automated auditing and formal property generation, identifying logic flaws that elude traditional tools [35]. DeepTx [25] combines multi-modal transaction features with LLM reasoning for real-time transaction risk analysis. Furthermore, research on automated repair, such as ACFix [57] and VulAdvisor [56], highlights the importance of guiding LLMs with mined patterns or local context to fix complex access control vulnerabilities. Augmenting prompts with semantic facts significantly boosts LLM performance over raw code [1], which motivates our ABT/APG-based prompt design.

7 Conclusion

We proposed TrapHunter, an end-to-end intent deviation detection framework for trap tokens. TrapHunter constructs a unified semantic representation using ABTs and APGs to normalize intra-procedural syntax and expose inter-procedural state dependencies, then applies LLM-powered reasoning to distinguish malicious intent, and finally confirms exploitability via fork-based dynamic validation. Evaluation on 269 real-world contracts (501 labeled samples) achieves an average precision of 81.8% and recall of 85.4%, significantly outperforming state-of-the-art tools, with generalization to 48 unseen on-chain contracts confirming robustness beyond curated benchmarks.

8 Data Availability

Our replication package is available online: https://doi.org/10.6084/m9.figshare.30082903

Acknowledgments

This research was supported by National Natural Science Foundation of China (62372367, 62232014, 62272377, 62372368), Shaanxi Province Sanqin Talent Introduction Program, the Singapore Ministry of Education Academic Research Fund Tier 2 (T2EP20224-0003) and the Nanyang Technological University Centre for Computational Technologies in Finance (NTU-CCTF). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of MOE and NTU-CCTF. The first author acknowledges the financial support from the China Scholarship Council.

References

[1] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. 2024. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th international conference on software engineering*. 1–13. doi:10.1145/3597503.3639183

[2] BlockSec. 2026. MetaSuites: The Swiss Army Knife for Builders. https://blocksec.com/metasuites. Accessed: January, 2026.

[3] Ramiro Camino, Christof Ferreira Torres, Mathis Baden, and Radu State. 2020. A data science approach for detecting honeypots in ethereum. In *2020 IEEE international conference on blockchain and cryptocurrency (ICBC)*. IEEE, 1–9. doi:10.1109/ICBC48266.2020.9169396

[4] Federico Cernera, Massimo La Morgia, Alessandro Mei, and Francesco Sassi. 2023. Token spammers, rug pulls, and sniper bots: An analysis of the ecosystem of tokens in ethereum and in the binance smart chain (BNB). In *32nd USENIX Security Symposium (USENIX Security 23)*. 3349–3366.

[5] Jiachi Chen, Jiang Hu, Xin Xia, David Lo, John Grundy, Zhipeng Gao, and Ting Chen. 2024. Angels or demons: investigating and detecting decentralized financial traps on ethereum smart contracts. *Automated Software Engineering* 31, 2 (2024), 63. doi:10.1007/s10515-024-00459-4

[6] Jiachi Chen, Mingyuan Huang, Zewei Lin, Peilin Zheng, and Zibin Zheng. 2025. To healthier ethereum: A comprehensive and iterative smart contract weakness enumeration. *Blockchain: Research and Applications* 6, 2 (2025), 100258. doi:10.1016/j.bcra.2024.100258

[7] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. 2019. Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 1503–1520. doi:10.1145/3319535.3345664

[8] Weili Chen, Xiongfeng Guo, Zhiguang Chen, Zibin Zheng, Yutong Lu, and Yin Li. 2020. Honeypot contract risk warning on ethereum smart contracts. In *2020 IEEE International Conference on Joint Cloud Computing*. IEEE, 1–8. doi:10.1109/JCC49151.2020.00009

[9] Wei Chen, Xinjun Jiang, Tian Lan, and Leyuan Liu. 2025. Ethereum fraud smart contract detection using heterogeneous semantic graph. *Automated Software Engineering* 32, 2 (2025), 1–19. doi:10.1007/s10515-025-00537-1

[10] ConsenSys Diligence. 2026. SÅńrya, The Sun God: A Solidity Inspector. https://github.com/ConsenSysDiligence/surya. Accessed: January, 2026.

[11] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024). doi:10.48550/arXiv.2412.19437

[12] De.Fi. 2026. REKT-Database. https://de.fi/rekt-database. Accessed: January, 2026.

[13] EthereumContractBackdoor. 2026. PiedPiperBackdoor: Ethereum Contract Backdoor. https://github.com/EthereumContractBackdoor/PiedPiperBackdoor. Accessed: January, 2026.

[14] Etherscan. 2026. Token Tracker (ERC-20). https://etherscan.io/tokens. Accessed: January, 2026.

[15] Tong Gu, Min Han, Songlin He, and Xiaotong Chen. 2023. Trap contract detection in blockchain with improved transformer. In *GLOBECOM 2023-2023 IEEE Global Communications Conference*. IEEE, 5141–5146. doi:10.1109/GLOBECOM54140.2023.10437216

[16] Maneesha Gunathilaka, Sandareka Wickramanayake, and HMN Dilum Bandara. 2024. DeFiTrust: A transformer-based framework for scam DeFi token detection using event logs and sentiment analysis. *Expert Systems with Applications* 251 (2024), 123913. doi:10.1016/j.eswa.2024.123913

[17] Kazuki Hara, Takeshi Takahashi, Motoya Ishimaki, and Kazumasa Omote. 2021. Machine-learning approach using solidity bytecode for smart-contract honeypot detection in the ethereum. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 652–659. doi:10.1109/QRS-C55045.2021.00099

[18] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. 2018. Kevm: A complete formal semantics of the ethereum virtual

machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 204–217. doi:10.1109/CSF.2018.00022

[19] Chao Hu, Yitian Chai, Hao Zhou, Fandong Meng, Jie Zhou, and Xiaodong Gu. 2024. How Effectively Do Code Language Models Understand Poor-Readability Code?. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 795–806. doi:10.1145/3691620.3695072

[20] Huiwen Hu, Qianlan Bai, and Yuedong Xu. 2022. Scsguard: Deep scam detection for ethereum smart contracts. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 1–6. doi:10.1109/INFOCOMWKSHPS54753.2022.9798296

[21] 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

[22] Han Liu, Daoyuan Wu, Yuqiang Sun, Haijun Wang, Kaixuan Li, Yang Liu, and Yixiang Chen. 2024. Using my functions should follow my checks: understanding and detecting insecure OpenZeppelin code in smart contracts. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 3585–3601.

[23] Yi Liu and Lizhi Cai. 2023. Honeytoken-Detector: A symbolic execution-based honeypot token detection tool. In *2023 26th ACIS International Winter Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD-Winter)*. IEEE, 134–139. doi:10.1109/SNPD-Winter57765.2023.10346207

[24] Yixuan Liu, Yuxin Dong, Ye Liu, Xiapu Luo, and Yi Li. 2025. Phantom Events: Demystifying the Issues of Log Forgery in Blockchain. *arXiv preprint arXiv:2502.13513* (2025). doi:10.48550/arXiv.2502.13513

[25] Yixuan Liu, Xinlei Li, and Yi Li. 2025. DeepTx: Real-Time Transaction Risk Analysis via Multi-Modal Features and LLM Reasoning. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 4025–4028. doi:10.1109/ASE63991.2025.00366

[26] Ye Liu, Yixuan Liu, Yi Li, and Cyrille Artho. 2025. Specification mining for smart contracts with trace slicing and predicate abstraction. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 147–158. doi:10.1109/SANER64311.2025.00022

[27] 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. In *Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS)*. Internet Society, San Diego, CA, USA. doi:10.14722/ndss.2025.241357

[28] Fuchen Ma, Meng Ren, Lerong Ouyang, Yuanliang Chen, Juan Zhu, Ting Chen, Yingli Zheng, Xiao Dai, Yu Jiang, and Jiaguang Sun. 2023. Pied-piper: Revealing the backdoor threats in ethereum erc token contracts. *ACM Transactions on Software Engineering and Methodology* 32, 3 (2023), 1–24. doi:10.1145/3560264

[29] Wei Ma, Daoyuan Wu, Yuqiang Sun, Tianwen Wang, Shangqing Liu, Jian Zhang, Yue Xue, and Yang Liu. 2025. Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 1742–1754. doi:10.1109/ICSE55347.2025.00027

[30] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023). doi:10.48550/arXiv.2303.08774

[31] OpenZeppelin. 2026. OpenZeppelin Docs. https://docs.openzeppelin.com/. Accessed: January, 2026.

[32] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*. IEEE, 1661–1677. doi:10.1109/SP40000.2020.00024

[33] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.

[34] Liya Su, Xinyue Shen, Xiangyu Du, Xiaojing Liao, XiaoFeng Wang, Luyi Xing, and Baoxu Liu. 2021. Evil under the sun: Understanding and discovering attacks on ethereum decentralized applications. In *30th USENIX Security Symposium (USENIX Security 21)*. 1307–1324.

[35] 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*. 1–13. doi:10.1145/3597503.3639117

[36] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023). doi:10.48550/arXiv.2312.11805

[37] Christof Ferreira Torres, Mathis Steichen, et al. 2019. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *28th USENIX Security Symposium (USENIX Security 19)*. 1591–1607.

[38] Fabian Vogelsteller and Vitalik Buterin. 2015. EIP-20: ERC-20 Token Standard. https://eips.ethereum.org/EIPS/eip-20. Accessed: January, 2026.

[39] Bin Wang, Xiaohan Yuan, Li Duan, Hongliang Ma, Chunhua Su, and Wei Wang. 2022. DeFiScanner: Spotting DeFi attacks exploiting logic vulnerabilities on blockchain. *IEEE Transactions on Computational Social Systems* 11, 2 (2022), 1577–1588. doi:10.1109/TCSS.2022.3228122

[40] Haijun Wang, Yurui Hu, Hao Wu, Dijun Liu, Chenyang Peng, Yin Wu, Ming Fan, and Ting Liu. 2024. Skyeye: Detecting imminent attacks via analyzing adversarial smart contracts. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1570–1582. doi:10.1145/3691620.3695526

[41] Haijun Wang, Yi Li, Shang-Wei Lin, Lei Ma, and Yang Liu. 2019. VULTRON: Catching vulnerable smart contracts once and for all. In *2019 IEEE/ACM 41st international conference on software engineering: New ideas and emerging results (ICSE-NIER)*. IEEE, 1–4. doi:10.1109/ICSE-NIER.2019.00009

[42] Haijun Wang, Ye Liu, Yi Li, Shang-Wei Lin, Cyrille Artho, Lei Ma, and Yang Liu. 2020. Oracle-supported dynamic exploit generation for smart contracts. *IEEE Transactions on Dependable and Secure Computing* 19, 3 (2020), 1795–1809. doi:10.1109/TDSC.2020.3037332

[43] Lijin Wang, Jingjing Wang, Tianshuo Cong, Xinlei He, Zhan Qin, and Xinyi Huang. 2025. From purity to peril: Backdooring merged models from âĂIJharmlessâĂİ benign components. In *USENIX Security Symposium (USENIX Security)*.

[44] Cong Wu, Jing Chen, Jian Shen, Guowen Xu, Yueming Wu, Haijun Wang, Hongwei Li, Yang Liu, and Yang Xiang. 2026. Catching Scam Tokens with Temporal Graph Learning in Decentralized Finance. *IEEE Transactions on Dependable and Secure Computing* (2026). doi:10.1109/TDSC.2026.3675905

[45] Cong Wu, Jing Chen, Ziming Zhao, Kun He, Guowen Xu, Yueming Wu, Haijun Wang, Hongwei Li, Yang Liu, and Yang Xiang. 2024. Tokenscout: Early detection of ethereum scam tokens via temporal graph learning. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 956–970. doi:10.1145/3658644.3690234

[46] Hao Wu, Haijun Wang, Shangwang Li, Yin Wu, Ming Fan, Wuxia Jin, and Ting Liu. 2025. RPHunter: Unveiling Rug Pull Schemes in Crypto Token via Code-and-Transaction Fusion Analysis. *arXiv preprint arXiv:2506.18398* (2025). doi:10.48550/arXiv.2506.18398

[47] Hao Wu, Haijun Wang, Shangwang Li, Yin Wu, Ming Fan, Ting Liu, and Xiapu Luo. 2026. Tracing the Shadows: Automatic Tracking and Analysis of Crypto Money Laundering via Transaction Semantic Analysis. *arXiv preprint arXiv:2607.18869* (2026). doi:10.48550/arXiv.2607.18869

[48] Yin Wu, Xiaofei Xie, Chenyang Peng, Dijun Liu, Hao Wu, Ming Fan, Ting Liu, and Haijun Wang. 2024. Advscanner: Generating adversarial smart contracts to exploit reentrancy vulnerabilities using llm and static analysis. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1019–1031. doi:10.1145/3691620.3695482

[49] Zhiying Wu, Jieli Liu, Jiajing Wu, Zibin Zheng, Xiapu Luo, and Ting Chen. 2023. Know your transactions: Real-time and generic transaction semantic representation on blockchain & web3 ecosystem. In *Proceedings of the ACM Web Conference 2023*. 1918–1927. doi:10.1145/3543507.3583537

[50] Zhiying Wu, Jiajing Wu, Hui Zhang, Zibin Zheng, and Weiqiang Wang. 2025. Hunting in the Dark Forest: A Pre-trained Model for On-chain Attack Transaction Detection in Web3. In *THE WEB CONFERENCE 2025*. doi:10.1145/3696410.3714928

[51] Pengcheng Xia, Haoyu Wang, Bingyu Gao, Weihang Su, Zhou Yu, Xiapu Luo, Chao Zhang, Xusheng Xiao, and Guoai Xu. 2021. Trade or trick? detecting and characterizing scam tokens on uniswap decentralized exchange. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 5, 3 (2021), 1–26. doi:10.1145/3491051

[52] Maoyi Xie, Ming Hu, Ziqiao Kong, Cen Zhang, Yebo Feng, Haijun Wang, Yue Xue, Hao Zhang, Ye Liu, and Yang Liu. 2024. Defort: Automatic detection and analysis of price manipulation attacks in defi applications. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 402–414. doi:10.1145/3650212.3652137

[53] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. 2022. xfuzz: Machine learning guided cross-contract fuzzing. *IEEE Transactions on Dependable and Secure Computing* 21, 2 (2022), 515–529. doi:10.1109/TDSC.2022.3182373

[54] Bosi Zhang, Ningyu He, Xiaohui Hu, Kai Ma, and Haoyu Wang. 2025. Following Devils' Footprint: Towards Real-time Detection of Price Manipulation Attacks. In *34th USENIX Security Symposium (USENIX Security 25)*. 4127–4145.

[55] Jiashuo Zhang, Jiachi Chen, Yiming Shen, Tao Zhang, Yanlin Wang, Ting Chen, Jianbo Gao, and Zhong Chen. 2025. When Crypto Fails: Demystifying Cryptographic Defects in Ethereum Smart Contracts. *IEEE Transactions on Software Engineering* (2025). doi:10.1109/TSE.2025.3551776

[56] Jian Zhang, Chong Wang, Anran Li, Wenhan Wang, Tianlin Li, and Yang Liu. 2024. Vuladvisor: Natural language suggestion generation for software vulnerability repair. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1932–1944. doi:10.1145/3691620.3695555

[57] Lyuye Zhang, Kaixuan Li, Kairan Sun, Daoyuan Wu, Ye Liu, Haoye Tian, and Yang Liu. 2025. ACF ix: Guiding LLMs with Mined Common RBAC Practices for Context-Aware Repair of Access Control Vulnerabilities in Smart Contracts. *IEEE Transactions on Software Engineering* (2025). doi:10.1109/TSE.2025.3590108

[58] Yichi Zhang, Zixi Liu, Yang Feng, and Baowen Xu. 2024. Leveraging large language model to assist detecting rust code comment inconsistency. In *Proceedings of the 39th IEEE/ACM international conference on automated software engineering*. 356–366. doi:10.1145/3691620.3695010

[59] Jianfei Zhou, Tianxing Jiang, Haijun Wang, Meng Wu, and Ting Chen. 2023. Dapphunter: Identifying inconsistent behaviors of blockchain-based decentralized applications. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 24–35. doi:10.1109/ICSE-SEIP58684.2023.00008

[60] Chenguang Zhu, Ye Liu, Xiuheng Wu, and Yi Li. 2022. Identifying solidity smart contract api documentation errors. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13. doi:10.1145/3551349.3556963

Received 2026-01-30; accepted 2026-06-25