EventSpec: Defining and Detecting Event-Semantic Issues in Blockchain Ecosystems

YIXUAN LIU, Nanyang Technological University, Singapore
YUXIN DONG, Peking University, China
YE LIU, Beijing Institute of Technology, China
YIN WU, Xi'an Jiaotong University, China
CHENGXUAN ZHANG, Nanyang Technological University, Singapore
XIAPU LUO, Hong Kong Polytechnic University, China
YI LI, Nanyang Technological University, Singapore

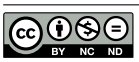
In recent years, smart contracts have become the backbone of decentralized applications (DApps), and off-chain systems such as bridges, wallets, and indexers rely heavily on event logs to track contract execution and state changes. However, the Ethereum Virtual Machine (EVM) does not validate or enforce event semantics, so logs can diverge from on-chain state, misleading off-chain systems into accepting incorrect state transitions. Existing smart contract vulnerability detection tools focus on logic bugs, with limited support for detecting event-semantic defects. To address this gap, we collect audit reports and incident cases and apply open card sorting to define five classes of event-semantic defects: event collision, state-event mismatch, unauthorized event emission, event emission mismatch, and event parameter mismatch. We propose EventSpec, which infers event specifications from a contract corpus via behavior inference and semantic-constraint extraction and applies differential checking to identify event-semantic defects in target contracts. We run EventSpec on 6,617 real-world contracts and evaluate detection effectiveness based on manually labeled results; EventSpec achieves an overall comprehensive precision of 90.17%. We further provide an off-chain evaluation harness that reproduces two off-chain attack vectors on any EVM-compatible chain: event origin confusion caused by unintended emitters and event–state desynchronization where events lack matching state updates. Using this harness, we demonstrate the feasibility of these attacks on bridge relayers, blockchain explorers, and NFT marketplaces, and report six wallet issues, four of which were confirmed (including a $600 bounty), with two remaining pending.

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

Additional Key Words and Phrases: Blockchain security, smart contract event, transaction logs

ACM Reference Format:
Yixuan Liu, Yuxin Dong, Ye Liu, Yin Wu, Chengxuan Zhang, Xiapu Luo, and Yi Li. 2026. EventSpec: Defining and Detecting Event-Semantic Issues in Blockchain Ecosystems. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA182 (October 2026), 24 pages. https://doi.org/10.1145/3832273

Authors' Contact Information: Yixuan Liu, Nanyang Technological University, Singapore, Singapore, liuy0255@e.ntu.edu.sg; Yuxin Dong, Peking University, Beijing, China, dongyuxin@stu.pku.edu.cn; Ye Liu, Beijing Institute of Technology, Beijing, China, ye.liu@bit.edu.cn; Yin Wu, Xi'an Jiaotong University, Xi'an, China, wuyin@stu.xjtu.edu.cn; Chengxuan Zhang, Nanyang Technological University, Singapore, Singapore, chengxua001@e.ntu.edu.sg; Xiapu Luo, Hong Kong Polytechnic University, Hong Kong, China, csxluo@comp.polyu.edu.hk; Yi Li, Nanyang Technological University, Singapore, Singapore, yi_li@ntu.edu.sg.



ACM 2994-970X/2026/10-ARTISSTA182
https://doi.org/10.1145/3832273

1 Introduction

Blockchain is a decentralized digital ledger technology that securely records and verifies transactions across multiple computers, ensuring data integrity and transparency [54]. Building on blockchain technology, Decentralized Applications (DApps) have emerged as a key innovation [11]. These applications run on smart contracts deployed across various blockchain networks, such as Ethereum, Polygon, and Binance Smart Chain (BSC). These contracts encode on-chain logic for managing state and digital assets, while a wide range of off-chain systems, such as bridges, wallets, explorers, and indexing services, interact with them to present user-facing state and functionality.

Smart contracts can emit **events**, which the EVM records on-chain as **transaction logs**. These logs provide a structured interface for off-chain consumption, and off-chain systems rely on them to track on-chain execution and maintain user-facing state. As of 2026-01-17, 97.9% of contract-involved successful Ethereum transactions emit at least one log [24], indicating that logs constitute a primary execution signal on Ethereum.

However, the EVM does not enforce the semantic correctness of logs. Smart contracts can emit events that omit, misstate, or imitate underlying state transitions. When off-chain systems treat logs as authoritative without validating emitters or checking contract state, misleading logs can trigger asset theft, fraud, and interoperability failures across applications.

Event-semantic issues arise when emitted events do not accurately summarize the underlying execution or when their origin and parameters are insufficiently validated. These issues include missing or extra emissions, parameter inconsistencies, events emitted under insufficient guards and so on. They are amplified by off-chain listeners assuming that logs from expected or known contracts imply valid state transitions, and that event parameters are safe to consume without validating the underlying state. Prior work only examined event misuse in specific domains, including cross-chain bridges [44, 75, 81], NFT ownership [37], code-documentation inconsistencies [83], and token behaviors [17]. However, a cross-application view of event semantics and their off-chain attack vectors remains underexplored. A unified taxonomy with detection methodology are still missing.

To close this gap, we conduct an empirical study of real-world incident reports and audit findings to derive a taxonomy of event-semantic defects and their associated off-chain attack vectors. Through open card sorting, we identify five types of event-semantic defects and abstract two recurring off-chain attack vectors that characterize how misleading logs propagate into off-chain systems. Based on this taxonomy, we develop EventSpec, a bytecode-oriented detector that constructs a corpus-derived event-emission specification and performs differential checking on target contracts. The specification captures both event layout properties (e.g., signatures and indexed positions) and semantic constraints (e.g., parameter origins and guard or binding signals), enabling EventSpec to flag deviations with explainable evidence, augmented by symbolic equality checks for identifying event-collision anomalies. In addition, EventSpec includes an off-chain evaluation harness for examining flaws in off-chain log consumption and validation.

We run EventSpec on 6,617 real-world contracts and evaluate detection effectiveness based on manually labeled results. EventSpec achieves an overall comprehensive precision of 90.17%. We further use the off-chain harness to reproduce the two attack vectors in a controlled environment and identify issues in bridge relayers, explorers, wallets, and NFT marketplaces. Finally, we discuss mitigation strategies for both smart contracts and off-chain systems to reduce such risks.

In short, we make the following contributions in this paper:

- **Event-Semantic Taxonomy.** We systematically analyze incidents and audit reports and define five types of event-semantic issues together with two off-chain attack vectors.

- **Corpus-Driven Detection.** We propose EventSpec, a bytecode-oriented framework that learns event and function semantic profiles from a corpus and performs differential checking on target contracts with explainable findings.
- **Empirical Evaluation.** We run EventSpec on 6,617 real-world contracts and evaluate detection effectiveness based on manually labeled results; EventSpec achieves an overall comprehensive precision of 90.17%.
- **Real-World Feasibility.** We implement an off-chain harness to assess the two attack vectors in real systems and report confirmed issues across bridge relayers, explorers, wallets, and NFT marketplaces (including a $600 bounty), alongside on-chain UEM findings.

2 Background

2.1 Smart Contract Events and Transaction Logs in EVM-Based Blockchains

Smart contracts, written in high-level languages such as Solidity [22] and Vyper [74], are compiled into bytecode and executed on the EVM [28]. EVM-based blockchains support a wide range of functionalities, from simple value transfers to complex decentralized applications (DApps) [1]. During execution, contracts may emit **events**, which the EVM records as **transaction logs** in the transaction receipt. Logs provide a structured format for off-chain consumption. Each log contains the address of the emitting contract, a set of topics, and a data field. For non-anonymous events, `topic0` stores the `keccak256` hash of the event signature, while *indexed* parameters are stored in subsequent topic slots and non-indexed parameters are stored in the data field. Logs are emitted via the `LOG0`–`LOG4` opcodes and are not part of the contract state, making them accessible only to off-chain consumers. For anonymous events, the event signature hash is omitted from `topic0`, and indexed parameters directly occupy the topic slots.

If a transaction reverts, all logs produced during its execution are discarded. A **transaction** is a user-submitted action that specifies a **sender** address, a target address, an optional value transfer, and input data encoding the invoked function and parameters. The sender signs the transaction, while the **emitter** refers to the contract that produces events during execution. Off-chain **listeners**, including DApps, wallets, and monitoring services, subscribe to events through RPC interfaces and process matching logs to react to on-chain activities. The EVM does not enforce semantic consistency between emitted logs and the underlying contract state, so off-chain systems must validate event emitters and relevant state transitions when correctness or security is required.

2.2 Token Standards

In EVM-based blockchains, tokens are implemented as smart contracts following standards like Ethereum Request for Comments 20 (*ERC-20*) and *ERC-721* [26, 73], which define interfaces and events to ensure DApp compatibility. For example, ERC-20 specifies `Transfer` and `Approval` events for tracking token transfers and approvals, while ERC-721 adopts similar events for unique NFTs. These ERC-defined events enable external systems like wallets and exchanges to monitor token activities in real time, updating user balances or ownership records.

2.3 Off-Chain Modules

Many DApps rely on off-chain components (e.g., relayers, indexers, and monitors) that observe on-chain transactions and events and react in real time. Cross-chain bridges are a typical case: events emitted on the source chain trigger off-chain processing and subsequent operations on the destination chain. In a standard design, a bridge contract emits a deposit event when assets are locked; an off-chain relayer monitors logs, extracts parameters, and invokes the destination-chain contract to mint or release assets. Events thus act as the primary coordination signal for cross-chain

Issue Name	FidoMeta – Missing functionality
Severity & Status	Critical & Fixed (`66f43e8`)
Description	Emits `Transfer` events without updating balances. **Location:** `FidoMeta.sol` (`_burn`, `_mint`, ctor)
Impact	May cause double-spending or incorrect reward accounting.

Bug Code Snippet	`function _mint(address account, uint256 amount) internal onlyOwner {`
	`  require(account != address(0));`
	`  require(totalSupply() + amount <= cap());`
	`  _total = _total.add(amount);`
	`  emit Transfer(address(0), account, amount);`
	`}`

Recommendation	Ensure consistency between state updates and emitted events.

Fig. 1. An example card generated from an event-semantic issue

transfers. If relayers accept logs without validating the emitter or corresponding state changes, misleading events can trigger incorrect off-chain actions. In practice, many listeners assume that logs from a trusted address uniquely represent the intended action and that event parameters accurately reflect state transitions, making direct state validation unnecessary. These assumptions improve performance but create the attack surface.

3 Understanding Event-Semantic Issues and Attack Vectors

We conduct an empirical study based on real-world security audit reports and attack incidents from EVM-compatible blockchains to define and classify event-semantic defects and their associated off-chain attack vectors.

3.1 Data Collection

3.1.1 Security Incidents. We collected publicly reported security incidents involving event-semantic defects by searching incident reports released by major blockchain security companies using the keywords *event* and *log*. The sources include public reports from SlowMist [64], BlockSec [7], PeckShield [59], and CertiK [13]. From these sources, we initially identified 46 security incidents.

3.1.2 Audit Reports. To complement incident data and cover a broader range of event-semantic defects, we analyzed smart contract audit reports collected by FORGE [15]. Starting from the full set of 6,454 audit reports, we applied keyword-based filtering to identify findings related to *event* and *log* usage, resulting in 1,354 security findings from 1,059 reports.

3.2 Data Analysis

3.2.1 Manual Filtering. To ensure relevance, two researchers independently reviewed all incidents and audit findings and resolved disagreements through discussion. Inter-rater agreement on the relevance/exclusion classification was measured using *Cohen's Kappa coefficient* (κ) [19], which quantifies inter-rater agreement corrected for chance; we obtained $\kappa = 0.89$, which under the Landis–Koch interpretation [41] ($\kappa \geq 0.81$) indicates *almost perfect* agreement. Disagreements were resolved by joint discussion, with two additional authors as adjudicators for unresolved cases. We excluded items not fundamentally tied to event semantics (e.g., generic phishing unrelated to event logs, front-end manipulation, key leakage, or unrelated bugs such as arithmetic errors or

Table 1. Definitions of the Five Event-Semantic Issues

Issue	Definition
Event Collision	Same event signature with overlapping parameter domains, obscuring origin.
State-Event Mismatch	Emitted logs are inconsistent with the on-chain state transition they claim to summarize.
Unauthorized Event Emission	Privileged events emitted without access control or checks.
Event Emission Mismatch	Event-emission behavior is wrong: missing, unexpected, extra, duplicated, misordered, or wrong-typed events.
Event Parameter Mismatch	The emitted event has the expected type, but its parameter schema, indexing, ordering, or value binding is wrong.

reentrancy), and retained cases where event usage, parameters, or event-state semantics directly affected off-chain interpretation or decision-making. All severity levels were retained, since event-semantic consequences depend on how off-chain consumers interpret the logs: a low-severity finding can still produce critical impact via a vulnerable bridge or wallet. The final dataset contains 1,020 audit reports with 1,294 findings and 11 security incidents, eight of which caused ~$115.6M in direct financial losses (e.g., Qubit Bridge $80M [5], pNetwork $13M [3], Meter Bridge $4.4M [4]), evidencing the substantial real-world harm caused by event-semantic defects, while the remaining three are event-driven phishing incidents (e.g., spoofed `Transfer` logs misleading wallets and explorers) with undisclosed losses.

3.2.2 Open Card Sorting. We applied open card sorting [66], consistent with prior empirical smart contract studies [47, 76]. Each confirmed incident or audit finding was converted into a card containing many fields, such as project name, issue description, and root cause captured in contract logic or code snippets. Two researchers, both with more than three years of smart-contract security research experience, conducted a two-round procedure: (i) jointly analyzing a random 40% sample to derive initial categories, and (ii) independently analyzing the remaining 60% using the derived categories, resolving disagreements through discussion. Inter-rater agreement on the independent 60% subset was $\kappa = 0.95$ (*almost perfect*), and disagreements were resolved using the same protocol as in Section 3.2.1. We excluded cards lacking sufficient evidence to attribute root causes (e.g., ambiguous contexts). Among the 1,294 retained audit findings, Event Emission Mismatch is the largest category (about 85%), reflecting that missing or redundant event emissions are the most prevalent event-semantic issues in smart contracts.

Figure 1 illustrates an example card generated from an event-semantic issue. In this case, several functions emit `Transfer` events without updating the corresponding state of user balances. Although the emitted events indicate token transfers, the underlying contract state remains unchanged. This mismatch between events and state can mislead off-chain systems that rely on events for accounting or validation, potentially resulting in incorrect balance tracking or double-spending risks. Based on root cause analysis, this issue is classified as a *State-Event Mismatch*.

3.3 Event-Semantic Issues in Smart Contracts

Our card-sorting analysis yielded five types of event-semantic defects, each defined in Table 1. To validate the taxonomy beyond inter-author agreement, we consulted three independent smart-contract security experts: a senior researcher with a dozen top-venue blockchain-security publications, a CertiK auditing partner with seven years of smart-contract auditing experience, and the founder of ExVul Security [30] with a decade of cybersecurity practice. After reviewing each category for meaningfulness, mutual exclusivity, and exhaustiveness, all three experts endorsed the five

```
1event Deposit(address indexed user, address indexed token, uint256 amount);
2function depositETH() external payable {
3    emit Deposit(msg.sender, address(0), msg.value);} // ETH is transferred via
         msg.value
4function deposit(address token, uint256 amount) external {
5    // missing: require(token != address(0)), returns false if token = zero
6    token.transferFrom(msg.sender, address(this), amount);
7    emit Deposit(msg.sender, token, amount);}
```

Fig. 2. An example of Event Collision.

```
1event Transfer(address indexed from, address indexed to, uint256 value);
2function transfer(address to, uint256 value) external {
3    emit Transfer(msg.sender, to, value); } // missing balance updates
```

Fig. 3. An example of State-Event Mismatch.

categories with no missing class identified, and contributed targeted refinements reflected in the per-category definitions.

3.3.1 Event Collision (EC). Event collision arises when an event signature is reused across distinct functions or execution paths with different semantics, where the emitted events have overlapping parameter domains. If off-chain systems only observe the event logs, the origin path is ambiguous and attackers can craft inputs that imitate another path, undermining event-based validation.
Example: Fig. 2 shows a simple bridge contract that emits a `Deposit` event from both the `depositETH` and `deposit` functions (Line 3 vs. Line 7). If `deposit` does not enforce `token != address(0)` (Line 5), an attacker can invoke `deposit` with `token = address(0)` to emit a `Deposit` event that is indistinguishable from a genuine ETH deposit without transferring any ETH. Off-chain validators that rely on `Deposit` logs can accept it and release assets on the destination chain; this class of event collision underlies incidents such as Qubit Bridge [5] and Meter Bridge [4], with losses of approximately $80 million and $4.4 million, respectively.

3.3.2 State-Event Mismatch (SEM). State–Event Mismatch occurs when an emitted event is inconsistent with the on-chain state transition it purports to summarize. For example, a contract may emit a `Transfer` event without the corresponding balance update, or log a value that differs from the updated state. Such mismatches violate the common off-chain assumption that events faithfully summarize state changes.
Example: Fig. 3 shows a contract that emits a `Transfer` event (Line 3) while omitting the corresponding balance updates for the sender and receiver. Off-chain systems that rely on `Transfer` logs will record a transfer that never occurred on-chain.

3.3.3 Unauthorized Event Emission (UEM). Unauthorized event emission refers to privileged events that untrusted callers can emit due to missing or insufficient access control, including cases where events are emitted before role or invariant checks. This misleads off-chain systems into inferring privileged actions that did not occur.
Example: The ConsenSys Diligence audit of DeFi Saver identified a missing access control check on the logging function `Log`; Line 3 shows an unrestricted emission. This can mislead off-chain monitors, as shown in Fig. 4 [21].

```
event LogEvent(address indexed _contract, address indexed _caller, bytes _data);
function Log(address _contract, address _caller, bytes memory _data) public {
    emit LogEvent(_contract, _caller, _data);} // no guard
```

Fig. 4. An example of Unauthorized Event Emission from the DeFi Saver V3 audit.

```
event Deposit(address indexed user, uint256 amount);
event Withdraw(address indexed user, uint256 amount);
function withdraw(uint256 amount) external {
    _balances[msg.sender] -= amount;
    emit Deposit(msg.sender, amount);} // mismatch event
function deposit() payable external {
    _balances[msg.sender] += msg.value;}
```

Fig. 5. An example of Event Emission Mismatch.

```
event Transfer(address indexed from, address indexed to, uint256 value);
function transfer(address to, uint256 value) external {
    _balances[msg.sender] -= value;
    _balances[to] += value;
    emit Transfer(to, msg.sender, value); // parameter order error
}
```

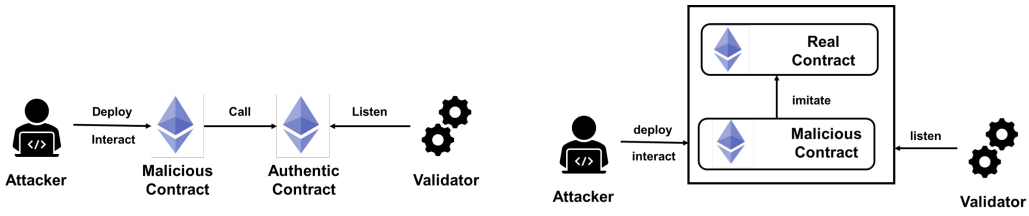
Fig. 6. An example of Event Parameter Mismatch (parameter error).

3.3.4 Event Emission Mismatch (EEM). Event Emission Mismatch concerns the event-emission behavior itself: the contract emits the wrong event type, omits an expected event, emits an unexpected or extra event, or emits the right event with an incorrect count or ordering. For example, a withdrawal function may emit `Deposit` instead of `Withdraw`, or a single logical action may produce duplicated events through multiple internal calls or alternative paths. These errors mislead off-chain observers about which actions actually occurred.
Example: As illustrated in Fig. 5, a withdrawal function emits a `Deposit` event instead of `Withdraw` (Line 5), so observers see an unexpected event and miss the expected one, leading off-chain systems to record incorrect balances or actions.

3.3.5 Event Parameter Mismatch (EPM). Event Parameter Mismatch occurs when an emitted event has the expected event type, but its parameters do not match the intended event schema or semantics. Examples include wrong parameter order (e.g., swapped `from`/`to` fields), incorrect indexed/non-indexed placement, or logging a stale or unrelated value. Such errors cause automated decoders and monitors to misinterpret the event.
Example: Fig. 6 shows a `Transfer` event where the `from` and `to` parameters are swapped (Line 5), resulting in an inverted transfer direction.

3.4 Off-Chain Attack Vectors

We abstract two recurring off-chain attack vectors. These vectors capture how misleading event semantics propagate into off-chain systems, such as validation systems, indexing explorers, and user-facing services. These vectors concern off-chain system behavior instead of contract-level defects, and emerge when event logs are accepted without validating the emitter or state.



(a) EOCA. (b) ESDA.

Fig. 7. Two off-chain attack vectors.

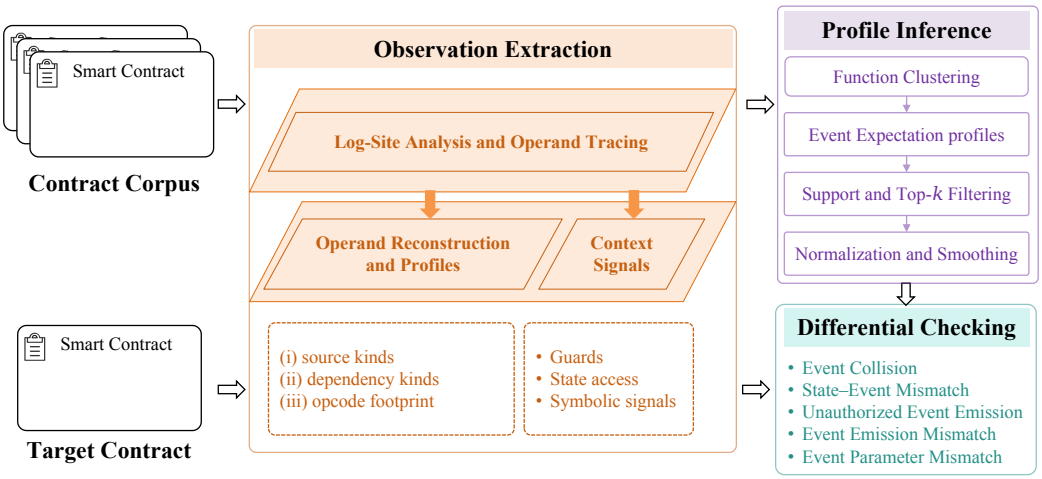


Fig. 8. Workflow of EventSpec.

3.4.1 Event Origin Confusion Attack (EOCA). Event Origin Confusion occurs when off-chain systems accept event logs from an unintended emitter, typically due to weak emitter validation or overly broad/incorrect whitelisting, especially when a single transaction contains logs from both a legitimate emitter and a malicious one.
Example: Fig. 7a illustrates this attack, and the pNetwork hack is a real-world case in which validators accepted events from an unintended contract, leading to unauthorized asset minting and theft of approximately $13M [3].

3.4.2 Event-State Desynchronization Attack (ESDA). This attack arises when off-chain systems infer behavior solely from events without validating on-chain state or storage updates.
Example: Fig. 7b shows this pattern, and the sleepminting incident is a representative case where forged NFT transfer events, despite no real ownership change, were consumed by NFT marketplaces that relied exclusively on event logs [37].

4 Methodology

This section presents EventSpec, which constructs an empirical event specification from a contract corpus and performs differential checking on target contracts to detect event-semantic defects, complemented by an off-chain testing module. EventSpec is corpus-driven, bytecode-oriented, and produces explainable findings.

4.1 Overview

EventSpec takes two inputs: (i) a contract corpus to learn typical event-emission behavior, and (ii) a target contract to be analyzed. Each contract bytecode is decompiled into three-address code (TAC); per-function control-flow graphs (CFGs) are built over TAC basic blocks, and an interprocedural control-flow graph (ICFG) links functions for bounded guard attribution. State-write bindings are computed intra-procedurally. As shown in Figure 8, the pipeline runs three modules: Extract captures observations at each log site, Infer aggregates them into event and function semantic profiles that form empirical event-emission specifications, and Diff compares target observations against these profiles. The output is a set of findings, each linked to a profile and supported by concrete evidence at the log and function levels.

4.2 Notation and Schema

We explicitly define an *event specification* as the set of event and function level profiles that summarize (i) layout information such as event signature hashes and indexed positions, (ii) emission behavior such as per-function emission frequency for each event type and log-shape distributions, and (iii) semantic constraints including parameter origins (argument/state/literal), guard predicates, and state-write bindings expected at emission sites. Each specification element is annotated with support and confidence statistics to enable confidence-gated checking.

Let $\mathcal{F}$ denote the set of externally callable functions and $\mathcal{L}$ the set of log sites in the analyzed contract. Each log site $\ell \in \mathcal{L}$ emits a `LOG` instruction with an event signature hash t_ℓ (i.e., `topic0`) and a label set L_ℓ over topics and data. Each emitted log is modeled as an *event observation* $o = \langle t, L, \Phi, G, W, B \rangle$, where t is the event signature hash, L is the label set over topics and data (e.g., `topic0..3`, `data0..n`), Φ captures per-label operand profiles, G summarizes access-control guards, W summarizes log-reachable state writes, and B captures operand–state binding evidence between event operands and reachable storage reads or writes.

For each event type t, let $\pi(t)$ denote the matched event profile inferred from the corpus. We treat $\pi(t)$ as an aggregated observation with the same fields (L, Φ, G, W, B) as a log-level observation. During differential checking, each log site ℓ is compared against its corresponding profile $\pi(t_\ell)$.

4.3 Observation Extraction

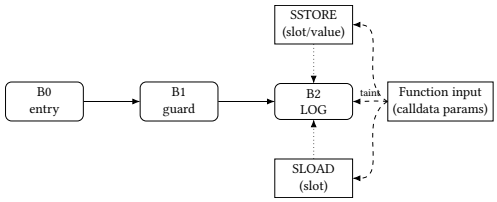
To instantiate event observations at each `LOG` site, we extract operand-level data-dependency signals and guard evidence from TAC and CFG structures, and summarize log-reachable state writes and bindings for the inference and differential checking modules.

4.3.1 Log-Site Analysis and Operand Tracing. To characterize each log site's operands and guards, we compute a log-reachable backward slice for each operand and derive an opcode footprint (Algorithm 1, Figure 9a). Separately, we identify dominator basic blocks for each log site as mandatory guard evidence used later to derive guard signals (Figure 9b).

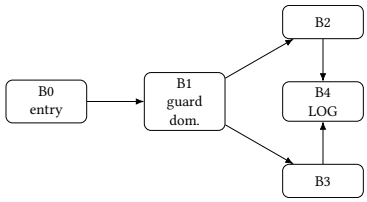
EventSpec first decompiles smart contract bytecode into TAC annotated with opcodes, `DefVars` and `UseVars` sets over TAC variables, and basic-block identifiers. As shown in Algorithm 1, operand-level opcode footprints are computed in three steps. First, EventSpec builds per-function CFGs and interprocedural control-flow graph (ICFG) connectivity, and computes reachability to each `LOG` site to restrict subsequent analysis to log-reachable statements (Lines 1–3). Second, for each `LOG` site ℓ and each of its operand labels λ, EventSpec constructs a log-reachable backward slice $Slice_\lambda$ over the CFG and initializes taint from the operand variables (Lines 4–9). Third, a slice-bounded backward taint analysis is applied over $Slice_\lambda$ to compute the taint closure T_λ, and the opcode footprint $\tau_{\ell,\lambda}$ is derived as the set of opcodes whose definitions or uses intersect T_λ within the slice

Algorithm 1: Log-Site CFG Slicing and Operand Tracing

Input: Bytecode **Output:** $\{\tau_{\ell,\lambda} \mid \ell \in \mathcal{L}, \lambda \in \mathcal{A}_\ell\}$
IR ← Decompile(Bytecode);
Build CFGs and ICFG with reachability; // for log-reachable slicing
$\mathcal{L} \leftarrow \{\ell \mid \ell$ is a LOG site$\}$;
foreach $\ell \in \mathcal{L}$ **do**
  $\mathcal{A}_\ell \leftarrow Operands(\ell)$; // topics & data
  foreach $\lambda \in \mathcal{A}_\ell$ **do**
    $Slice_\lambda \leftarrow BackSlice(\lambda, \text{CFG})$; // log-reachable slice subgraph
    $T_\lambda \leftarrow Vars(\lambda)$; // taint starts from the operand (sink)
    $T_\lambda \leftarrow \textsc{TaintAnalysis}(Slice_\lambda, T_\lambda)$; // slice-based taint closure
    $\tau_{\ell,\lambda} \leftarrow \{op(v) \mid v \in Slice_\lambda \wedge (\text{DefVars}(v) \cup \text{UseVars}(v)) \cap T_\lambda \neq \emptyset\}$; // tainted opcode footprint
  end
end
Procedure *TaintAnalysis*(S, T)**:**
  $T_{old} \leftarrow \emptyset$;
  while $T \neq T_{old}$ **do**
    $T_{old} \leftarrow T$;
    foreach $v \in S$ **do**
      if $\text{DefVars}(v) \cap T \neq \emptyset$ **then**
        $T \leftarrow T \cup \text{UseVars}(v)$; // backward taint step
      end
    end
  end
  return T;
return $\{\tau_{\ell,\lambda} \mid \ell \in \mathcal{L}, \lambda \in \mathcal{A}_\ell\}$;



(a) CFG slice around a log site.



(b) Dominator-based guard analysis.

Fig. 9. Log-site analysis: CFG slicing/operand tracing (left) and dominator-based guard inference (right).

(Lines 10–25). The resulting opcode footprint $\tau_{\ell,\lambda}$ serves as an operand-level feature in Φ and is used by the profile inference and differential checking modules.

To conservatively capture operand origins under bounded analysis, we treat function-input sources from calldata (e.g., CALLDATALOAD/ARG), environment sources (e.g., CALLER, CALLVALUE, ORIGIN), and memory/storage reads (e.g., MLOAD, SLOAD) as taint sources when deriving operand source/dependency kinds. Dependencies are propagated along def-use chains within the log-reachable slice.

Figure 9a illustrates the log-reachable slice subgraph produced by Algorithm 1. For each operand, we trace its data dependencies within this slice and summarize the derivation as an opcode footprint, providing a lightweight signature without full path exploration.

Figure 9b illustrates dominator-based guard inference. For a log site in basic block b_ℓ, any basic block d that dominates b_ℓ must be executed on all paths that reach the log site; therefore, predicates

in d form mandatory emission conditions (e.g., in the figure, $B1$ dominates the log block). We extract guard evidence from such dominator blocks (e.g., `require` checks or branch conditions) and map them to guard categories, which are also used by the inference and differential checking modules.

4.3.2 Operand Reconstruction and Profiles. Each log site ℓ has shape $s_\ell = (n_t, n_d)$,where n_t is the number of topics (indexed operands) and n_d is the number of data words (non-indexed operands).. Indexed operands can read from the topic region. In contrast, non-indexed operands are not directly accessible and must be reconstructed from the data region by locating relevant `MSTORE` writes in the current basic block and its predecessors, resolving pointer arithmetic, and aligning offsets to 32-byte slots. Each aligned slot is assigned a data label (`data0`, `data1`, …) to represent reconstructed non-indexed operand. Auxiliary attributes such as literal class (`zero`/`nonzero`) and argument index (when operand originates from calldata params, i.e., `ARG`/`CALLDATALOAD`) are derived to refine operand profiles during matching and comparison. For each operand label λ, operand profile includes: (*i*) source kinds (*ii*) dependency kinds (e.g., `arith`, `bitwise`, `compare`, `hash`, `mem`, `storage`, `data`, `env`, `call`) and (*iii*) opcode footprint from the def-use path. *Lightweight constraint* from the def-use path, including comparisons against constants, nonzero checks, masking or shifts.

4.3.3 Context Signals. Contextual signals are captured at each log site and explicitly mapped to guard validation, state–event consistency checking, and event collision detection. For **guards**, we classify categories by opcode-structure heuristics over dominator blocks, aligned with common permission-constraint patterns summarized by PrettySmart [82], including owner checks, role-based checks, whitelist lookups, and data-driven checks; when an internal function containing a log site lacks local guards, caller functions are searched via a bounded interprocedural traversal, and guard categories are attributed from the call chain, since internal helpers are not directly callable and their effective guards are enforced at external entry points in $\mathcal{F}$. For **state updates and reads**, we restrict `SSTORE` statements to those in the same function and on CFG paths that reach the basic block containing the log site with statement order before the log, summarize slot classes (constant / linear / mapping-like) inferred from slot-expression forms, record two binding modes for writes (*value-binding* when an operand path intersects the `SSTORE` value path, and *slot-binding* when it intersects the `SSTORE` slot path), and record storage-read bindings when an operand path intersects the slot path of a reachable `SLOAD`; these bindings are used to flag state-event mismatches, including inputs emitted directly from calldata parameters (`ARG`/`CALLDATA`) without any binding to reachable storage reads/writes. For **signatures and symbolic signals**, t_0 is matched to the known event signature when available and the hash remains the primary key for matching and inference; path constraints and symbolic expressions are extracted from the taint slice and guard predicates, and SMT solving is applied at comparison to assign each operand pair an equality status (`possible` for satisfiable, `unsat` for provably unequal), with unknown results discarded.

4.4 Profile Inference

4.4.1 Function Clustering. Functions are clustered by a behavioral signature. We define $\mathrm{sig}(f) = \mathrm{hash}(G_f, S_f, D_f, M_f, O_f)$, where G_f is the guard-category set, S_f the storage-write kinds, D_f the frequent operand dependency/source kinds, M_f the modal (topic count, data count) pair across log sites, and O_f the frequent opcodes. Functions with the same signature form a cluster c.

4.4.2 Event Expectations and Shape Profiles. At the cluster level, the expected probability of emitting a given event type is estimated as $P_c(t_0) = n_c(t_0)/|F_c|$, where $n_c(t_0)$ counts functions in cluster c that emit t_0 and F_c is the set of functions in c. The distribution of total log counts and per-topic log counts, which enables checks for missing or over-emitted events, is included. For each t_0, we use a log-shape distribution over (n_t, n_d), where n_t and n_d denote the topic and data label counts,

together with the indexed-label set (e.g., `topic1..`) to capture indexed/data layout expectations; deviations from the dominant shape are treated as potential parameter mismatches. Other per-label operand, guard, and binding profiles are aggregated from counts and used in similarity scoring.

4.4.3 Support and Top-k Filtering. Support denotes the number of contributing functions for function-semantic profiles and the number of event observations for event-semantic profiles (per cluster and t_0). To control noise for common events, only the top-k event-semantic profiles per t_0 by support are retained, and differential checking enforces a minimum support threshold.

4.4.4 Normalization and Smoothing. Before differential checking, frequency profiles are normalized into probability distributions. For distributional similarities we add an α pseudo-count to each label before normalization, and for Bernoulli-style presence checks we clamp probabilities using the same α to avoid 0/1 extremes [16]. This makes similarity scores stable even when a profile has limited support and prevents a single missing label from collapsing the overall score.

4.5 Differential Checking

Differential checking compares a target contract against the corpus-derived specification. We match extracted events and functions to their closest inferred profiles, score similarity over the corresponding feature sets, and apply confidence gating (minimum similarity, minimum support, and Wilson-interval constraints) before recording taxonomy-aligned findings with evidence. In parallel, we run target-only checks that do not depend on profile matching, including missing-log detection, event collision checks, and insufficient-guard checks.

4.5.1 Profile Matching and Similarity. For a target contract, observations are extracted and each event observation is matched to the best event profile for its t_0, and each function is matched to the best function profile. Event similarity is a weighted combination of indexed-label likelihood, parameter-count likelihood, operand source similarity, operand dependency similarity, `SSTORE`-kind overlap, and guard-category overlap:

$$\mathrm{sim}_e = \frac{\sum_k w_k \, \mathrm{sim}_k}{\sum_k w_k}$$

Indexed-label similarity uses the average log-likelihood of label presence under the indexed profile, and count similarity uses the probability of the observed topic/data counts. Operand source/dependency similarities use Jensen-Shannon similarity [45] over per-label distributions, and guard/SSTORE overlap is measured by Jaccard similarity. Function similarity is computed as $\mathrm{sim}_f = w'_{st}\,\mathrm{sim}_{st} + w'_{gd}\,\mathrm{sim}_{gd} + w'_{op}\,\mathrm{sim}_{op}$, where sim_{st} and sim_{gd} are Jaccard overlaps over storage-write kinds and guard categories, and sim_{op} is the Jaccard overlap of high-frequency opcodes; the weights w'_* sum to 1.

4.5.2 Confidence Gating and Thresholds. Findings are emitted only when similarity exceeds a minimum matching threshold and support exceeds a minimum support count. To avoid brittle decisions on small samples, each probability-based decision is gated with a Wilson confidence [77]. For an estimated probability p computed from n supporting samples, the interval width is:

$$w = \frac{2z}{1 + z^2/n}\sqrt{\frac{p(1-p)}{n} + \frac{z^2}{4n^2}}$$

We require $w \le w_{\max}$, where z is the normal critical value for the chosen confidence level and $w_{\max}$ is the maximum allowed width. Guard and storage checks have their own thresholds, and operand-profile checks use label-specific thresholds for source/literal/argument index. All probability computations here use the smoothed distributions described in Sect. 4.4.4.

Field group	Profile $\pi(t_\ell)$	Target o at log site ℓ
Layout	t_ℓ $(n_t, n_d) = (3, 1)$ $L = \{\text{topic0..2, data0}\}$	t_ℓ $(n_t, n_d) = (3, 1)$ $L = \{\text{topic0..2, data0}\}$
Operands	Φ: src(topic1)=CONST footprint: …	Φ: src(topic1)=CONST footprint: …
Guards	G: owner (dominant)	G: none
Constraints	B: data0 ↔ `SSTORE.value` …	B: …

Fig. 10. Example: missing guard evidence.

Table 2. Detection rules for differential finding categories.

Finding	Detection rule
Event Collision	$\exists \ell_1 \neq \ell_2 : t_{\ell_1} = t_{\ell_2} \wedge s_{\ell_1} = s_{\ell_2} \wedge L_{\ell_1} = L_{\ell_2} \wedge \text{SAT}(\text{Eq}(o_{\ell_1}, o_{\ell_2}))$
State–Event Mismatch	$\exists \ell \in \mathcal{L}, \exists \lambda \in (L_\ell \cap L_{\pi(t_\ell)}) : \text{Bound}_\ell(\lambda) \neq \text{Bound}_{\pi(t_\ell)}(\lambda)$
Unauthorized Event Emission	$\exists f \in \mathcal{F}, \exists \ell \in \mathcal{L} : \ell \in f \wedge G_f = \emptyset \wedge G_{\pi(f)} \neq \emptyset$
Event Emission Mismatch	$\exists f \in \mathcal{F}, \exists t_0 : \text{Count}_f(t_0) \neq \lvert\{\ell \in \mathcal{L} \mid \ell \in f \wedge t_\ell = t_0\}\rvert$
Event Parameter Mismatch	$\exists \ell \in \mathcal{L} : (s_\ell \neq s_{\pi(t_\ell)} \vee L_\ell \neq L_{\pi(t_\ell)}) \vee \exists \lambda \in (L_\ell \cap L_{\pi(t_\ell)}) : \Phi_\ell[\lambda] \neq \Phi_{\pi(t_\ell)}[\lambda]$

4.5.3 Findings. The Diff module produces findings aligned with the taxonomy in Table 1. Each finding is instantiated by comparing a target observation against its matched profile and applying a category-specific detection rule that combines profile-derived expectations with target-side consistency checks, including SMT-backed signals.

Figure 10 illustrates the evidence used by the differential checker. Each target log site is compared against its matched event profile $\pi(t_\ell)$, and each target function is compared against its matched function profile to derive expected emission behavior. The example highlights missing guard evidence, which triggers an Unauthorized Event Emission finding when the log-emitting function has an empty guard summary.

For an observation x, we denote its label set, log shape, and operand profiles as L_x, s_x, and Φ_x, respectively. We use $\text{Eq}(\cdot)$ to denote the conjunction of SMT-encodable operand-equality constraints and $\text{SAT}(\cdot)$ to denote satisfiable constraints. For binding evidence, $\text{Bound}_x(\lambda)$ indicates that label λ has any operand–state binding recorded via log-reachable storage reads or writes.

We classify findings by applying the detection rules summarized in Table 2. For function-level comparisons, we write $\text{Count}_f(t_0)$ for the expected emission count of event type t_0 under the matched function profile of a target function f, subject to the confidence and thresholding constraints. In brief, "Event Collision" flags indistinguishable emissions whose operand constraints overlap; "State–Event Mismatch" flags disagreement between expected and observed operand–state binding evidence; "Unauthorized Event Emission" flags log-emitting functions lacking any guard evidence after interprocedural attribution; "Event Emission Mismatch" flags deviations between expected and observed per-function event counts; and "Event Parameter Mismatch" flags inconsistencies in log layout or per-label operand profiles relative to the matched event profile.

4.6 Off-Chain Evaluation Harness

The harness is an *evaluation* component, not part of the static analysis pipeline. Its purpose is to test whether off-chain consumers (wallets, explorers, bridge relayers) accept misleading event semantics that an on-chain contract could legitimately emit. We design two attacker-side test contracts.

EOCA test contract. In a single transaction, the contract calls the legitimate target so that a genuine event is emitted from the target's address, and emits a forged event of the same type from a separate attacker-controlled emitter address. An off-chain consumer that allowlists by event type alone, without binding each event to its intended emitter, will accept the forged event as genuine.

ESDA test contract. The contract emits an event (e.g., `Transfer`) from an attacker-controlled address *without* performing the corresponding state update (e.g., balance change). An off-chain consumer that infers state purely from event logs, without cross-checking the on-chain state, will record a state change that never happened (the sleepminting pattern).

5 Evaluation

5.1 Research Questions

In this section, we aim to answer the following research questions.

- **RQ1:** How ERC-consistent are corpus-inferred event-emission specifications?
- **RQ2:** How accurate is EventSpec and how common are event-semantic issues in the wild?
- **RQ3:** How feasible is it to perform the two off-chain attack vectors in real-world systems?

Implementation. EventSpec uses Gigahorse [34, 40] to decompile bytecode into TAC and construct CFGs, and symbolic equality checks are supported by the Greed module [63]. The remaining components (extraction, inference, and differential checking) are implemented in about 7k lines of Python, and the harness includes 51 lines of Solidity.

Configuration. We evaluate EventSpec in a controlled environment on a workstation with 4 physical CPU cores (8 threads), 64 GB RAM, and Ubuntu 22.04.5 LTS. We deduplicate contracts by bytecode hash. Default settings: smoothing α=1.0, min support 20, top-k event profiles 3, event similarity threshold 0.65, function similarity threshold 0.65, guard/binding threshold 0.8, confidence level z=1.96, and max interprocedural depth 6. We also set a per-contract analysis time limit of 1200 s, a path exploration depth limit of 500, and an SMT solver timeout of 300 s.

5.2 Datasets

Data source. We derive our datasets from the Smart Contract Sanctuary collection of verified Etherscan contracts [58]. We use the snapshot that includes contracts verified up to July 13, 2023, and selected contract files deployed on the Ethereum mainnet, totaling 331,382 mainnet contracts.

Corpus (behavior inference). We deduplicate contracts by deployed bytecode and exclude contracts that exceed the 150 s decompilation timeout, yielding 212,072 contracts.

Target (detection pool). We use 6,617 verified Ethereum smart contracts as the target pool for large-scale detection (RQ2), drawn from the NumScout-curated set [14] over Smart Contract Sanctuary [58]. The set retains contracts with more than 100 historical transactions, a non-zero ether balance, and successful compilation under the recorded Solidity version, focusing the evaluation on mainstream deployed Ethereum smart contracts that are substantially larger and more modern than common benchmarks such as SmartBugs [32] (e.g., 11.5× LOC, 5.5× instructions, and a larger fraction compiled with Solidity $\geq$ 0.8.0). The 95-contract manually labeled dataset described below is a random subset of this target pool, not an additional pool.

ERC standards dataset. For ERC events, we derive event specifications directly from ERC documents listed on the EIPs site [27]. We include all ERC standards published on the Ethereum Improvement Proposals site up to Jan. 20, 2026 with Final status. To focus on adopted standards, we

Table 3. RQ1 sensitivity to inference parameters.

Parameter	Value	Profiles	Signature match	Layout match	Semantic match
α	0.1	1246	80.77%	100.00%	90.48%
	0.5	1246	80.77%	100.00%	90.48%
	1	1246	80.77%	100.00%	90.48%
	2	1246	80.77%	100.00%	90.48%
	5	1246	80.77%	100.00%	90.48%
Min support	5	8510	100.0%	96.15%	88.46%
	10	2954	96.15%	96.00%	88.00%
	20	1246	80.77%	100.00%	90.48%
Top-k	1	793	80.77%	80.95%	71.43%
	3	1246	80.77%	100.00%	90.48%
	5	1439	80.77%	100.00%	90.48%
	10	1695	80.77%	100.00%	95.24%

Table 4. RQ1 sensitivity to the ERC inclusion threshold.

Min contracts	Events	Signature match	Layout match	Semantic match
50	26	80.77%	100.00%	90.48%
100	22	95.45%	100.00%	90.48%
250	15	100.0%	100.00%	86.67%
1000	12	100.0%	100.00%	83.33%

map ERCs to implementations using the Smart Contract Sanctuary dataset and exclude ERCs with fewer than 50 contracts to ensure sufficient support, yielding 14 ERCs and 26 event definitions [58]. We extract event declarations to obtain layout, and annotate normative statements to derive semantic constraints on parameter origins and required guards. Two authors annotate normative statements and resolve disagreements through discussion.

Manually labeled dataset. We manually label contracts randomly sampled from the filtered dataset for the presence of the five issue types. Following the sampling protocol used in concurrent peer-reviewed work [14, 47, 65], applying Cochran's formula with finite-population correction at 95% confidence and a ±10% margin of error gives $n = 95$ for the pool of 6,617 contracts. Two authors jointly label 30% of the samples to establish labeling criteria, then independently label the remaining 70% and resolve disagreements through discussion. Inter-rater agreement on the independently labelled 70% subset was $\kappa = 0.97$ (*almost perfect*), and any remaining labelling disputes were arbitrated by two additional authors. This labeled subset serves as the ground truth for the per-category detection accuracy reported in RQ2.

5.3 RQ1: ERC Consistency of Inferred Specifications

RQ1 provides a controlled validation setting for our corpus-driven inference engine: it evaluates whether specifications inferred from real-world implementations recover ERC event declarations, the only authoritative reference for Ethereum events. We therefore compare inferred specifications against ERC-declared layouts and normative semantic constraints before applying empirical inference to events that lack formal specifications. We evaluate corpus-inferred event-emission specifications for layout and semantic consistency against ERC event declarations and normative

statements. We decompose the specifications into two components: (i) layout and (ii) a semantic profile. Layout is defined as the event signature and indexed parameter positions, and the semantic profile is defined by parameter origin constraints (argument-tainted, state-tainted, or unconstrained) and required guard/binding conditions. A layout match requires the correct signature in `topic0` and indexed positions. A semantic match requires that the inferred origin and guard/binding signals satisfy all documented constraints. We report layout/semantic match rates for ERC events with sufficient support for inference. For ERC events with sparse support, we do not attempt to infer or evaluate semantics due to insufficient evidence.

Match rate definitions. Let E be the set of evaluated ERC events and $E_{\text{sig}} \subseteq E$ the subset whose inferred `topic0` matches the ERC-declared signature. The signature match rate is $|E_{\text{sig}}|/|E|$. Layout and semantic match rates are computed over E_{sig}, as $|E_{\text{layout}}|/|E_{\text{sig}}|$ and $|E_{\text{semantic}}|/|E_{\text{sig}}|$, respectively, where E_{layout} and E_{semantic} additionally satisfy indexed-position and semantic constraints.

RQ1 summary and parameter effects. On the ERC standards dataset, Table 3 shows that varying α does not materially change layout or semantic matching. This is because α only smooths empirical frequencies, and for ERC events with enough observations the dominant layout and origin/guard signals already exceed decision thresholds. The same table also shows that signature non-matches under the default setting are mainly due to limited ERC adoption after bytecode deduplication: some ERC standards have few remaining implementations and fail the support threshold, while lowering support allows these signatures to be inferred. With top-k=1, layout match rates drop, indicating that the most common real-world usage of an event is not always the ERC-required one, and ERC-compliant behavior can appear as a lower-support mode that is pruned when only the single dominant pattern is kept.

Implications and configuration choice. Table 4 shows that excluding low-adoption ERCs improves signature and layout agreement, but shows no consistent improvement in semantic agreement, which can decrease as the threshold increases. This indicates that the main limitation lies in semantic inference rather than data sparsity. In particular, many ERC documents specify authorization or state-update conditions that do not align with the limited guard types we currently infer, which also differ from the five guard categories studied in PrettySmart [82]. Based on these observations, we use the default setting for RQ2 (α=1, minimum support = 20, top-k=3), which gives stable inference, avoids low-support noise, and preserves ERC-compliant patterns that are not the single most frequent mode. Taken together, the high layout-level agreement supports the inference methodology, while semantic-level disagreements show that real-world implementation patterns do not always conform to ERC specifications, motivating empirical inference for events without formal specifications.

5.4 RQ2: Detection Accuracy and Prevalence

We estimate real-world prevalence over the complete target pool of 6,617 contracts, counting each issue type at most once per contract. Detection effectiveness is evaluated on the 95 manually labeled contracts from the same pool; five additional Event Collision cases collected during open card sorting are used only for Event Collision effectiveness.

We report precision/recall/F1 per issue type and an overall comprehensive precision. Metrics are computed at the contract level: a contract is positive for an issue if any log site exhibits it, and a detection is correct if EventSpec reports at least one matching finding for that issue in the contract. Precision/recall/F1 follow standard definitions. We compute comprehensive precision as $P_{\text{comp}} = \frac{\sum_i P_i \cdot |C_i|}{\sum_i |C_i|}$, where $|C_i|$=TP_i+FN_i, and report macro-averaged recall across issue types.

Table 5 summarizes RQ2 results, reporting per-issue metrics as well as the overall comprehensive precision. Overall, the comprehensive precision reaches 90.17%, and the macro-average recall is

Table 5. RQ2 results: large-scale detection yield on the target pool of 6,617 contracts, and detection effectiveness on 95 labeled contracts (Event Collision uses five open card sorting contracts).

Issue	#	Pct.(%)	TP	FP	FN	Precision(%)	Recall(%)	F1-score(%)
Event Collision	0	0	5	0	0	100.00	100.00	100.00
State-Event Mismatch	4159	62.85	45	9	0	83.33	100.00	90.91
Unauthorized Event Emission	405	6.12	11	2	1	84.62	91.67	88.00
Event Emission Mismatch	6514	98.44	90	1	1	98.90	98.90	98.90
Event Parameter Mismatch	3664	55.37	18	9	2	66.67	90.00	76.60

96.11%. State-Event Mismatch is frequently caused by smart contracts that emit events before updating state, which violates common best practices; we treat such cases as true positives. False positives for State-Event Mismatch and Event Parameter Mismatch mainly arise when the inferred semantic profiles of smart contracts differ across event, parameter, or state dimensions, despite representing the same underlying intent. For EPM specifically, the lower precision mainly stems from semantic ambiguity at the parameter level: equivalent event semantics can be implemented using different parameter representations (e.g., derived values such as scaled storage reads, reordered fields, or indirect state references through helper functions), and our inference matches syntactic patterns rather than semantic equivalence. For Unauthorized Event Emission, false positives mainly stem from incomplete permission inference, while false negatives are primarily caused by decompilation failures that prevent accurate recovery of function blocks. Event Emission Mismatch errors are largely due to limited inter-contract call analysis: although the current contract does not emit the expected event, the emission occurs elsewhere along the call chain. In general, issues associated with explicit guards or binding signals (e.g., Unauthorized Event Emission and State-Event Mismatch) tend to be easier to detect, whereas issues that rely on parameter-origin inference (e.g., Event Parameter Mismatch) remain more challenging.

Overall, at least 6,514 out of 6,617 contracts (98.44%) contain at least one tool-reported issue, since Event Emission Mismatch alone appears in 6,514 contracts. The most prevalent issue type is Event Emission Mismatch (98.44%), followed by State-Event Mismatch (62.85%) and Event Parameter Mismatch (55.37%). Unauthorized Event Emission appears in 6.12% of contracts. Although Event Collision is rare in our datasets, it has led to real-world attacks and measurable losses in cross-chain applications. The high EEM yield is dominated by `no_event_emission`, EventSpec's operational rule for non-view functions with no reachable log but with state changes or other non-view operations. Thus, 98.44% denotes findings under the detector definition rather than independently confirmed exploitable vulnerabilities in 98.44% of contracts.

Comparison with General-Purpose Smart Contract Analyzers. To assess whether general-purpose smart-contract analyzers cover event-semantic defects, we run four widely used tools on the same 95 labelled contracts, spanning complementary analysis paradigms: Solidity-AST data-flow analysis (Slither [31]), industry-oriented SWC-based symbolic execution (Mythril [20]), XPath/rule-based pattern matching (SmartCheck [69]), and academic symbolic execution (Oyente [51]). We conservatively map each tool's documented detectors to our five categories, retaining a detector only when its documented semantics directly match one of our categories; ambiguous or non-event detectors are excluded rather than force-classified.

Under this mapping, only three Slither detectors qualify: `events-maths` and `events-access` detect missing events after arithmetic or access-control changes, covering only a subset of EEM, while `reentrancy-events` detects reentrancy-induced event-state desynchronization, covering

Table 6. Tool comparison: per-category F1 (%) on the 95 labelled contracts.

Tool	EC	SEM	UEM	EEM	EPM	Avg
EventSpec	100.00	90.91	88.00	98.90	76.60	90.88
Slither [31]	–	20.55	–	69.07	–	17.92

a subset of SEM. Mythril, SmartCheck, and Oyente do not provide detectors designed for event-emission paths and therefore achieve 0% F1 across all five categories. Accordingly, Table 6 reports per-category F1 for EventSpec and Slither, the only baseline with non-zero F1. For Slither's two non-zero categories, SEM has low precision (17.74%) and recall (24.44%), while EEM has perfect precision (100.00%) but limited recall (52.75%). The low SEM precision is consistent with prior evidence that smart-contract SAST tools, including detectors such as Slither's `reentrancy-events`, exhibit high false-positive rates [42].

Slither's coverage is also limited to narrow SEM/EEM sub-cases: it misses wrong event types, parameter swaps, event-signature collisions, and missing emissions unrelated to arithmetic or access-control changes. These results indicate that event-semantic defect detection remains a design gap in general-purpose smart-contract analyzers, rather than merely a matter of detector tuning.

5.5 Real-World Attack Feasibility (RQ3)

Targets. We study two target surfaces. *Off-chain targets:* log consumers, including blockchain explorers, wallets, NFT marketplaces, indexers, and bridge relayers. These are selected from (1) previously reported sleepminting incidents [37], (2) wallet programs listed on bug-bounty platforms such as BugRap [10], and (3) widely used blockchain explorers (e.g., BscScan, OKLink, Bitquery, Tokenview, and Bsctrace) [6, 8, 9, 56, 70]. For EOCA on bridges, we additionally search open-source bridge projects and historically affected bridge relayers with public audit reports (e.g., Sygma relayer audits) [67]. *On-chain targets:* deployed contracts whose events are used as off-chain triggers, drawn from a set of 100 active contracts listed on the Dune dashboard "How Users React" [25].

Methodology. We separate targets into two groups: (i) off-chain consumers (e.g., explorers, wallets, relayers) for testing the two attack vectors and (ii) on-chain contracts for UEM auditing. For off-chain targets, we collect test addresses/contracts and derive candidate scenarios for the two attack vectors. Using our off-chain harness, we craft and send test transactions (on a forked local chain or testnet) and then verify whether emitted logs and state transitions satisfy emitter constraints, event signatures, indexed topics, and state deltas. We also observe downstream behaviors to determine whether the events are accepted or misinterpreted. When public evidence is incomplete, we reproduce the workflow in a controlled environment or replay historical attack transactions, avoiding any impact on live systems. For on-chain targets, we run EventSpec over the 100 active contracts from the Dune dashboard to identify unauthorized event emission and manually validate candidates. We responsibly report confirmed cases.

Result. (1) EOCA. We classify EOCA as cases where off-chain consumers accept events from unintended emitters due to insufficient emitter validation, and we confirm emitting contracts differ from legitimate project addresses. We identify some bridge-relayer pipelines vulnerable to emitter confusion; we reproduce one historically affected bridge relayer and confirm a newly affected open-source bridge relayer implementation that has been forked and deployed, with reports filed and acknowledged [23, 60, 67]. **(2) ESDA.** We classify the following cases as ESDA because the off-chain consumer infers transfers purely from event logs without validating corresponding state changes, and we confirm missing balance/ownership deltas on-chain: (a) We observe transfer-spoofing transactions that emit transfer logs without corresponding token state changes, causing apparent

movements from `0x8888...8888` to a victim wallet; despite no actual token movement, at least five major explorers (BscScan, OKLink, Bitquery, Tokenview, and Bsctrace) display the spoofed event as legitimate [6, 8, 9, 56, 70]; (b) During audits, we found ESDA issues in six cryptocurrency wallets, reported them, and received four confirmations, including a $600 bounty; two reports remain pending; (c) We identified a log-display issue in Blockscout that can mislead users about event data [2]; (d) We design a bypass to reproduce the sleepminting pattern on testnets for two major NFT marketplaces (OpenSea [57] and Rarible [61]). **(3) UEM.** In our on-chain auditing, we find three DeFi projects and one GameFi project vulnerable to unauthorized event emission (privileged events emitted without proper access control); the largest affected project had a market capitalization of $169,688 as of Oct. 15, 2024, based on Etherscan [29].

Finding: Insufficient event semantic validation creates on-chain/off-chain inconsistencies. Our results reveal three root causes: (1) off-chain services that neglect emitter validation can be tricked into processing events from attacker-controlled contracts, enabling unauthorized operations; (2) off-chain services that omit state-delta checks display spoofed transfers as legitimate, misleading users about actual token holdings; and (3) contracts lacking access control for privileged events allow attackers to emit arbitrary logs, causing off-chain services to record fabricated data that diverges from true on-chain state.

5.6 Threats to Validity

Internal validity. (1) *Data extraction and labelling:* Manual filtering, taxonomy construction, and ground-truth labelling may involve subjective judgement. We mitigate this through independent double labelling, inter-rater agreement reporting, disagreement resolution, and validation by three industry smart-contract security experts. (2) *Severity treatment:* We do not stratify findings by severity because impact depends on off-chain log interpretation; a low-severity audit finding can have critical impact when consumed by a vulnerable bridge or wallet. This affects severity analysis only. (3) *Detection accuracy:* Detection accuracy is bounded by decompilation failures and solver/time limits. We manually cross-validated all 95 labelled samples to catch decompilation-induced false negatives at labelling time. (4) *Baseline comparability:* Because general-purpose analyzers are not designed for event semantics, we conservatively map detectors to categories and exclude non-event detectors.

External validity. Generalisability depends on how well the contract corpus and taxonomy-source dataset represent real-world practice. (1) *Data representativeness:* The contract corpus is drawn from Smart Contract Sanctuary [58], whereas the taxonomy-source dataset covers FORGE-derived [15] audit findings and confirmed security incidents reported through the end of 2025. Public incident reports and audit findings may still underrepresent rare or undocumented event-semantic issues. (2) *Evaluation scope:* Our evaluation targets active contracts on a pool of 6,617 contracts, and off-chain case studies cover bridges, explorers, wallets, and marketplaces; results may not generalise to inactive or highly specialised contracts. (3) *Temporal stability:* Finalised ERC event signatures are immutable after standardisation, so inferred specifications remain applicable to newer contracts that implement those standards. (4) *Corpus independence:* The EventSpec pipeline does not hard-code corpus-specific assumptions, so it can be re-run on any sufficiently large bytecode corpus to produce updated specifications without changing the methodology, taxonomy, or detection rules.

6 Related Work

6.1 Smart Contract Security Analysis

Smart-contract analyzers span several paradigms. **Static analysis** includes XML/XPath rules in SmartCheck [69], bytecode decompilation in Gigahorse [35], source-level rules in Slither [31], AST

analysis in NeuCheck [50], and bytecode taint tracking in eTainter [33]. **Symbolic execution** ranges from Oyente and source-level SolSEE [46, 51] to inter-contract or slicing-based systems such as Pluto, CRUSH, and JACKAL [36, 52, 62]; Mythril and Manticore are widely used implementations [20, 53]. **Dynamic analysis** includes ContractFuzzer, ConFuzzius, and Smartian [18, 38, 71], while **formal verification** systems such as Zeus, Securify, and Maian check specified properties [39, 49, 55, 72].

These systems target general vulnerability templates or user-supplied properties, including reentrancy, access control, arithmetic, storage, and call-flow properties, rather than event semantics. Event-semantic checking requires a reference for emitter, timing, and parameter–state consistency. Formal approaches require that reference in advance, whereas EventSpec infers event and function profiles from deployed bytecode and differentially checks targets, including non-ERC events and conventions without written specifications.

6.2 Smart Contract Events

Prior work studies event usage and gas-inefficient emissions [43], cross-chain security events and vulnerabilities [44, 75, 78, 81], log-based token tracking [12, 48], sleepminting [37, 79], code–documentation–event inconsistencies [83], and ERC-20 behavioral mismatches [17]. These studies focus on particular domains, while off-chain infrastructure still reconstructs state from logs and ABI expectations without necessarily validating emitter or on-chain state. Our work provides a cross-domain taxonomy of event-semantic defects and recurring off-chain attack vectors that explicitly covers emitter and state–event consistency.

7 Discussion and Mitigation

Event-semantic issues arise from both contract logic and off-chain interpretation. At the **contract level**, developers should emit events after state updates, guard privileged emissions, avoid ambiguous signature reuse, validate ambiguous inputs (e.g., zero-address versus native-token semantics), and keep parameter order and indexing consistent with the ABI. At the **off-chain level**, consumers should allowlist emitters; validate signatures, topic counts, parameter types, and reverted status; and confirm state or call context for high-risk transfers, minting, and withdrawals. Consumers should also bind each event to its intended call and validate the complete sequence (count, order, emitter, and call path), while monitoring log signals against state invariants and versioned schemas.

Quantitative overhead of on-chain state verification. We traced 26 Ethereum mainnet transactions from four Xscope cross-chain incidents (THORChain #1/#2/#3 and Qubit Bridge) involving Inconsistent Event Parsing or Unrestricted Deposit Emitting [80]. For each, we called Tenderly's `debug_traceTransaction` with the `prestateTracer` (diffMode) three times and averaged the round-trip latency [68]. In deployment, address allowlists and event signatures can filter candidates before tracing.

Across all 26 transactions, the median trace latency is 900 ms (average 1381 ms, P95 3030 ms, max 7238 ms). All traces complete well within Ethereum's 12 s block budget; the slowest uses 60% of it. The tracer returns pre/post storage differences for checking event–state correspondence, and a local archive node can remove network round trips. The results support transaction tracing for real-time verification.

8 Conclusion

EventSpec detects event-semantic defects through corpus-inferred event and function profiles and explainable differential checking. We define five defect types and two off-chain attack vectors. Across 6,617 contracts and labeled data, EventSpec achieves 90.17% comprehensive precision. Its harness reproduces both vectors and confirms issues in relayers, explorers, wallets, and NFT marketplaces. We also provide mitigations for contracts and off-chain consumers.

Data Availability

Our replication package is available online at https://github.com/yxsec/eventspec.

Acknowledgments

We thank the following three external experts for their valuable feedback in reviewing and validating our taxonomy: Peiyu Wang from CertiK, Weibo Wang from ExVul Security Inc., and Dr. Jianzhong Su, a Research Fellow at Nanyang Technological University. This research is supported by the Singapore Ministry of Education Academic Research Fund Tier 2 (T2EP20224-0003), Academic Research Fund Tier 1 (RG12/23), 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.

References

[1] Andreas M Antonopoulos and Gavin Wood. 2018. *Mastering Ethereum: Building Smart Contracts and Dapps*. O'reilly Media.

[2] EventSpec Authors. 2024. Error in displaying msg.sender in Transaction Logs. https://github.com/blockscout/blockscout/issues/9879.

[3] Rob Behnke. 2021. EXPLAINED: THE PNETWORK HACK (SEPTEMBER 2021). https://www.halborn.com/blog/post/explained-the-pnetwork-hack-september-2021.

[4] Rob Behnke. 2022. EXPLAINED: THE METER.IO HACK (FEBRUARY 2022). https://www.halborn.com/blog/post/explained-the-meter-io-hack-february-2022.

[5] Rob Behnke. 2022. EXPLAINED: THE QUBIT HACK (JANUARY 2022). https://www.halborn.com/blog/post/explained-the-qubit-hack-january-2022.

[6] Bitquery. 2026. Bitquery. https://explorer.bitquery.io/.

[7] BlockSec. 2026. BlockSec. https://blocksec.com/.

[8] Bscscan. 2026. BNB Smart Chain (BNB) Blockchain Explorer. https://bscscan.com/.

[9] Bsctrace. 2026. Bsctrace. https://bsctrace.com/.

[10] BugRap. 2026. BugRap Bug Bounty Platform. https://bugrap.io/.

[11] Vitalik Buterin. 2014. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper/. White paper.

[12] 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.

[13] CertiK. 2026. CertiK. https://www.certik.com/.

[14] Jiachi Chen, Zhenzhe Shao, Shuo Yang, Yiming Shen, Yanlin Wang, Ting Chen, Zhenyu Shan, and Zibin Zheng. 2025. NumScout: Unveiling Numerical Defects in Smart Contracts using LLM-Pruning Symbolic Execution. *IEEE Transactions on Software Engineering* (2025). doi:10.1109/TSE.2025.3555622

[15] Jiachi Chen, Yiming Shen, Jiashuo Zhang, Zihao Li, John Grundy, Zhenzhe Shao, Yanlin Wang, Jiashui Wang, Ting Chen, and Zibin Zheng. 2026. FORGE: An LLM-driven Framework for Large-Scale Smart Contract Vulnerability Dataset Construction. In *Proceedings of the 48th International Conference on Software Engineering (ICSE)*.

[16] Stanley F. Chen and Joshua Goodman. 1999. An Empirical Study of Smoothing Techniques for Language Modeling. *Computer Speech & Language* 13, 4 (1999), 359–394. doi:10.1006/csla.1999.0128

[17] 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 (CCS)*. 1503–1520. doi:10.1145/3319535.3345664

[18] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 227–239. doi:10.1109/ASE51524.2021.9678888

[19] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. doi:10.1177/001316446002000104

[20] Consensys. 2025. Mythril. https://github.com/ConsenSys/mythril/.

[21] ConsenSys Diligence. 2021. DeFi Saver V3 Audit Report, March 2021. https://diligence.security/audits/2021/03/defi-saver/.

[22] Chris Dannen. 2017. *Introducing Ethereum and Solidity*. Vol. 1. Springer. doi:10.1007/978-1-4842-2535-6

[23] dl tokene. 2026. bridge-core Repository. https://github.com/dl-tokene/bridge-core.

[24] Dune. 2026. Dune Data Catalog: Ethereum Raw Transactions. https://docs.dune.com/data-catalog/evm/ethereum/raw/transactions.

[25] Dune. 2026. How Users React. https://dune.com/kimichi/how-users-react.

[26] William Entriken, Dieter Shirley, Jacob Evans, and Nastassia Sachs. 2018. EIP-721: ERC-721 Non-Fungible Token Standard. https://eips.ethereum.org/EIPS/eip-721.

[27] Ethereum Improvement Proposals Editors. 2026. Ethereum Improvement Proposals. https://eips.ethereum.org/.

[28] Ethereum.org. 2025. Ethereum Virtual Machine (EVM). https://ethereum.org/en/developers/docs/evm/.

[29] Etherscan. 2026. Etherscan. https://etherscan.io/.

[30] ExVul Security. 2026. ExVul Security. https://exvul.com/.

[31] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15. doi:10.1109/WETSEB.2019.00008

[32] João F Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2020. SmartBugs: A Framework to Analyze Solidity Smart Contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1349–1352. doi:10.1145/3324884.3415298

[33] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2022. eTainter: detecting gas-related vulnerabilities in smart contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 728–739. doi:10.1145/3533767.3534378

[34] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. IEEE, 1176–1186. doi:10.1109/ICSE.2019.00120

[35] Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. 2022. Elipmoc: Advanced Decompilation of Ethereum Smart Contracts. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–27. doi:10.1145/3527321

[36] Fabio Gritti, Nicola Ruaro, Robert McLaughlin, Priyanka Bose, Dipanjan Das, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. 2023. Confusum Contractum: Confused Deputy Vulnerabilities in Ethereum Smart Contracts. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1793–1810.

[37] Barbara Guidi and Andrea Michienzi. 2022. Sleepminting, the brand new frontier of Non Fungible Tokens fraud. In *Proceedings of the 2022 ACM Conference on Information Technology for Social Good (GoodIT)*. 75–81. doi:10.1145/3524458.3547239

[38] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 259–269. doi:10.1145/3238147.3238177

[39] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society. doi:10.14722/ndss.2018.23082

[40] Sifis Lagouvardos, Yannis Bollanos, Neville Grech, and Yannis Smaragdakis. 2025. The Incredible Shrinking Context... in a Decompiler Near You. *Proceedings of the ACM on Software Engineering (ISSTA)* (2025). arXiv:2409.11157. doi:10.1145/3728935

[41] J. Richard Landis and Gary G. Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* 33, 1 (1977), 159–174. doi:10.2307/2529310

[42] Kaixuan Li, Yue Xue, Sen Chen, Han Liu, Kairan Sun, Ming Hu, Haijun Wang, Yang Liu, and Yixiang Chen. 2024. Static Application Security Testing (SAST) Tools for Smart Contracts: How Far Are We? *Proc. ACM Softw. Eng.* 1, FSE, Article 65 (July 2024), 24 pages. doi:10.1145/3660772

[43] Lantian Li, Yejian Liang, Zhihao Liu, and Zhongxing Yu. 2023. Understanding Solidity Event Logging Practices in the Wild. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*. 300–312. doi:10.1145/3611643.3616342

[44] Zeqin Liao, Yuhong Nan, Henglong Liang, Sicheng Hao, Juan Zhai, Jiajing Wu, and Zibin Zheng. 2024. SmartAxe: Detecting Cross-Chain Vulnerabilities in Bridge Smart Contracts via Fine-Grained Static Analysis. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 249–270. doi:10.1145/3643738

[45] Jianhua Lin. 1991. Divergence measures based on the Shannon entropy. *IEEE Transactions on Information Theory* 37, 1 (1991), 145–151. doi:10.1109/18.61115

[46] Shang-Wei Lin, Palina Tolmach, Ye Liu, and Yi Li. 2022. SolSEE: A Source-Level Symbolic Execution Engine for Solidity. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*. Association for Computing Machinery, New York, NY, USA, 1687–1691. doi:10.1145/3540250.3558923

[47] Zewei Lin, Jiachi Chen, Jingwen Zhang, Zexu Wang, Yuming Feng, Weizhe Zhang, and Zibin Zheng. 2025. SSR: Safeguarding Staking Rewards by Defining and Detecting Logical Defects in DeFi Staking. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, 571–583. doi:10.1109/ASE63991.2025.00054

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

[49] 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)*. 147–158. doi:10.1109/SANER64311.2025.00022

[50] Ning Lu, Bin Wang, Yongxin Zhang, Wenbo Shi, and Christian Esposito. 2021. NeuCheck: A more practical Ethereum smart contract security analysis tool. *Software: Practice and Experience* 51, 10 (2021), 2065–2084. doi:10.1002/spe.2745

[51] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security (CCS)*. 254–269. doi:10.1145/2976749.2978309

[52] Fuchen Ma, Zhenyang Xu, Meng Ren, Zijing Yin, Yuanliang Chen, Lei Qiao, Bin Gu, Huizhong Li, Yu Jiang, and Jiaguang Sun. 2021. Pluto: Exposing vulnerabilities in inter-contract scenarios. *IEEE Transactions on Software Engineering* 48, 11 (2021), 4380–4396. doi:10.1109/TSE.2021.3117966

[53] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1186–1189. doi:10.1109/ASE.2019.00133

[54] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system.

[55] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th annual computer security applications conference (ACSAC)*. 653–663. doi:10.1145/3274694.3274743

[56] Oklink. 2026. OKLINK. https://www.oklink.com/.

[57] OpenSea. 2026. OpenSea NFT Marketplace. https://opensea.io/.

[58] Martin Ortner and Shayan Eskandari. 2024. Smart Contract Sanctuary. https://github.com/tintinweb/smart-contract-sanctuary

[59] PeckShield. 2026. PeckShield. https://peckshield.com/.

[60] QtumProject. 2026. bridge-core Repository. https://github.com/qtumproject/bridge-core.

[61] Rarible. 2026. Rarible NFT Marketplace. https://rarible.com/.

[62] Nicola Ruaro, Fabio Gritti, Robert McLaughlin, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. 2024. Not your Type! Detecting Storage Collision Vulnerabilities in Ethereum Smart Contracts. In *Netw. Distrib. Syst. Security Symp.* 1–17. doi:10.14722/ndss.2024.24713

[63] Nicola Ruaro, Fabio Gritti, Robert McLaughlin, Dongyu Meng, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. 2025. A History of Greed: Practical Symbolic Execution for Ethereum Smart Contracts. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 275–296. doi:10.1007/978-3-031-97623-0_17

[64] SlowMist. 2026. SlowMist. https://www.slowmist.com/.

[65] Hao Song, Teng Li, Jiachi Chen, Ting Chen, Beibei Li, Zhangyan Lin, Yi Lu, Pan Li, and Xihan Zhou. 2025. Enhancing The Open Network: Definition and Automated Detection of Smart Contract Defects. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 1281–1293. doi:10.1109/ICSE55347.2025.00119

[66] Donna Spencer. 2009. *Card Sorting: Designing Usable Categories.* Rosenfeld Media.

[67] SprinterTech. 2026. Sygma Relayer Audits. https://github.com/sprintertech/sygma-relayer/tree/main/audits.

[68] Tenderly. 2026. Tenderly: Web3 Development Platform with Monitoring and Alerts. https://tenderly.co/.

[69] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 9–16. doi:10.1145/3194113.3194115

[70] Tokenview. 2026. Tokenview. https://bsc.tokenview.io/.

[71] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. CONFUZZIUS: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 103–119. doi:10.1109/EuroSP51992.2021.00018

[72] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC conference on computer and*

communications security (CCS). 67–82. doi:10.1145/3243734.3243780

[73] Fabian Vogelsteller and Vitalik Buterin. 2015. EIP-20: ERC-20 Token Standard. https://eips.ethereum.org/EIPS/eip-20.

[74] Vyperlang. 2026. Vyper. https://github.com/vyperlang/vyper.

[75] Ke Wang, Yue Li, Che Wang, Jianbo Gao, Zhi Guan, and Zhong Chen. 2024. Xguard: Detecting inconsistency behaviors of crosschain bridges. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE)*. 612–616. doi:10.1145/3663529.3663809

[76] Zexu Wang, Jiachi Chen, Tao Zhang, Yu Zhang, Weizhe Zhang, Yuming Feng, and Zibin Zheng. 2025. Copy-and-Paste? Identifying EVM-Inequivalent Code Smells in Multi-chain Reuse Contracts. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1031–1053. doi:10.1145/3728921

[77] Edwin B Wilson. 1927. Probable inference, the law of succession, and statistical inference. *J. Amer. Statist. Assoc.* 22, 158 (1927), 209–212. doi:10.1080/01621459.1927.10502953

[78] 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. https://arxiv.org/abs/2607.18753. arXiv:2607.18753 [cs.SE]

[79] Lei Xiao, Shuo Yang, Wen Chen, and Zibin Zheng. 2025. WakeMint: Detecting Sleepminting Vulnerabilities in NFT Smart Contracts. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 740–750. doi:10.1109/SANER64311.2025.00075

[80] Xscope-Tool. 2024. Xscope: Hunting for Cross-Chain Bridge Attacks — Public Detection Results. https://github.com/Xscope-Tool/Results.

[81] J. Zhang, J. Gao, Y. Li, Z. Chen, Z. Guan, and Z. Chen. 2022. Xscope: Hunting for cross-chain bridge attacks. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1–4. doi:10.1145/3551349.3559520

[82] Zhijie Zhong, Zibin Zheng, Hong-Ning Dai, Qing Xue, Junjia Chen, and Yuhong Nan. 2024. Prettysmart: Detecting permission re-delegation vulnerability for token behaviors in smart contracts. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*. 1–12. doi:10.1145/3597503.3639140

[83] 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 (ASE)*. 1–13. doi:10.1145/3551349.3556963

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