

From Transactions to Exploits: Automated PoC Synthesis for Real-World DeFi Attacks

Xing Su¹, Hao Wu¹, Hanzhong Liang¹, Yunlin Jiang¹, Yuxi Cheng¹, Yating Liu¹, Fengyuan Xu^{1,*}

¹State Key Laboratory for Novel Software Technology, Nanjing University
China

xingsu@smail.nju.edu.cn, hao.wu@nju.edu.cn

{hanz_liang, 231220040, yuxicheng, yatingliu}@smail.nju.edu.cn
fengyuan.xu@nju.edu.cn

Abstract

Blockchain systems are increasingly targeted by on-chain attacks that exploit contract vulnerabilities to extract value rapidly and stealthily, making systematic analysis and reproduction highly challenging. In practice, reproducing such attacks requires manually crafting proofs-of-concept (PoCs), a labor-intensive process that demands substantial expertise and scales poorly. In this work, we present the *first* automated framework for synthesizing verifiable PoCs directly from on-chain attack executions. Our key insight is that attacker logic can be recovered from low-level transaction traces via trace-driven reverse engineering, and then translated into executable exploits by leveraging the code-generation capabilities of large language models (LLMs). To this end, we propose TRACEXP, which localizes attack-relevant execution contexts from noisy, multi-contract traces and introduces a novel dual-decompiler to transform concrete executions into semantically enriched exploit pseudocode. Guided by this representation, TRACEXP synthesizes PoCs and refines them to preserve exploitability-relevant semantics. We evaluate TRACEXP on 321 real-world attacks over the past 20 months. TRACEXP successfully synthesizes PoCs for 93% of incidents, with 58.78% being directly verifiable, at an average cost of only \$0.07 per case. Moreover, TRACEXP enabled the release of a large number of previously unavailable PoCs to the community, earning a \$900 bounty and demonstrating strong practical impact.

1 Introduction

The decentralized finance (DeFi) ecosystem has experienced explosive growth in recent years, enabling complex financial primitives such as flash loans, decentralized exchanges, and permissionless lending protocols. As of January 2026, DeFi protocols collectively manage nearly \$130 billion in total value locked (TVL) [11]. At the core of this prosperity are smart contracts, autonomous programs that directly control and transfer substantial on-chain assets without intermediaries. While this design enables transparency and composability, it also amplifies security risks: any flaw in business logic design or implementation can be exploited with scalable losses [45, 82]. In 2025 alone, on-chain attacks caused losses exceeding \$2.9 billion [50], with many incidents involving complex interactions across multiple contracts and transactions.

Effectively responding to such incidents requires more than detecting attacks [43, 79], it critically depends on *understanding how exploits were carried out* in sufficient detail to support incident response, root-cause analysis, and the design of effective defenses.

In practice, this level of understanding is most reliably achieved through proof-of-concept (PoC), which provide a precise, verifiable, and end-to-end representation of attack behavior. While manually curated PoCs remain labor-intensive and require expert knowledge, community efforts such as DeFiHACKLABS [10] have made significant contributions by collecting hundreds of accessible PoCs written by security experts for real-world attacks. Nevertheless, our measurement of wild incidents indicates that ~50% of attacks still lack PoCs, and as the number and complexity of incidents continue to grow, this gap is likely to widen, highlighting the urgent need for scalable and automated approaches.

Fortunately, recent developments provide a practical opportunity to achieve this automation. On-chain transactions record *concrete, path-sensitive* executions of real-world exploits, capturing the exact information (e.g., fund flows) during an attack. Moreover, decades of progress in EVM reverse engineering have yielded decompilers [9, 39, 42, 49, 58] that lift low-level bytecode into structured, human-readable representations, offering a potential bridge from raw traces to higher-level program logic. In parallel, recent advances in LLMs have demonstrated strong capabilities in code generation [31, 33]. Together, these developments suggest a promising direction: *automatically synthesizing PoCs by transforming execution evidence into concise and reproducible exploit logic*.

However, turning this opportunity into a practical system poses non-trivial challenges. Execution traces are low-level, verbose, and span multiple contracts (e.g., attacker contracts, victim protocols, and token contracts), making it difficult to isolate attack-relevant logic directly. While existing decompilers offer structured views of EVM bytecode, they are fundamentally designed for whole-program analysis and decoupled from transaction-level execution context. Besides, they struggle to faithfully capture dynamic behaviors critical to exploits, such as call arguments, data flows, and fund flows [48, 75, 77]. Finally, synthesizing *verifiable* PoCs is inherently attack-specific. Although LLMs can generate source code, their hallucinations [52, 80] prevent guarantees that the PoCs are reproduce the original exploit, making validation essential.

To address these challenges, we propose TRACEXP, the *first* automated framework for synthesizing verifiable PoCs directly from on-chain attack transactions. TRACEXP first localizes attack-relevant execution contexts from noisy, multi-contract traces and lifts them into a compact, contract-centric representation. It then introduces a *trace-driven dual decompiler* that integrates static decompilation with dynamic values observed during execution, enabling deterministic recovery of high-level exploit pseudocode from a single concrete trace. Guided by this representation, TRACEXP leverages

*Corresponding author.

LLMs for PoC synthesis and employs an *exploit-aware refinement loop* that validates reproduced exploits via fund-flow oracles, prioritizing semantic exploitability over strict trace equivalence.

We evaluate TRACExp on 321 real-world DeFi attacks collected over 20 months. TRACExp successfully synthesizes PoCs for 93% of these incidents, with roughly half being directly verifiable, at an average runtime <5 minutes and monetary cost of ~\$0.07 per case. Beyond controlled evaluation, TRACExp enabled the contribution of 33 previously unavailable PoCs to the community within 2 days, accounting for 38% of all submissions during that month (ranked 1st) and earning \$900 bounty. Measured per attack event, this output substantially exceeds that of experts, demonstrating markedly higher efficiency and effectiveness than manual PoC construction.

In summary, we make the following contributions:

- **New Framework.** We propose TRACExp, the *first* general framework that automatically synthesizes verifiable PoC directly from on-chain attack transactions. TRACExp is attack-agnostic and requires no prior knowledge of vulnerability types or source code, enabling robust reproduction across diverse real-world attacks.
- **New Method.** We introduce a *trace-driven reverse engineering* technique that integrates static decompilation with dynamic values from attack traces, enabling a *dual decompiler* to deterministically recover concise, high-level exploit pseudocode from a single execution, overcoming imprecision that limit static analysis.
- **New Validation Mechanism.** We design an *exploit-aware validation and refinement mechanism* that leverages fund-flow oracles to ensure semantic exploitability, while automatically repairing syntactic and semantic deviations in synthesized PoCs, guaranteeing they are both compilable and faithful to the original exploit.
- **Large-Scale Evaluation.** We conduct one of the *largest* real-world on-chain attack reproductions. TRACExp successfully synthesizes PoCs for 93% of evaluated incidents at an average monetary cost of \$0.07 per case, with 58.78% of them being directly verifiable.
- **Real-World Impact.** We demonstrate the practical utility by contributing 33 previously unavailable PoCs to the community within 2 days, showing that automated PoC synthesis can surpass expert-driven efforts in both efficiency and effectiveness.

2 Background and Motivation

In this section, we first provide some background knowledge on blockchain and DeFi attacks (§ 2.1), and then we introduce existing DeFi attacks analysis techniques and discuss their limitations (§ 2.2).

2.1 A Primer on Blockchain and DeFi Attacks

Blockchain and Smart Contracts. Blockchains are decentralized, tamper-resistant ledgers recording *transactions*. Platforms like Ethereum and BNB Smart Chain extend this model with *smart contracts*, self-executing programs that automate trustless interactions. Transactions transfer value, invoke contract functions, or deploy code, and originate from either *externally owned accounts* (EOAs)

or *contract accounts*. *Tokens*, standardized via protocols like ERC-20 [13] and ERC-721 [14], and stablecoins (e.g., USDT [61]) provide liquidity and underpin most on-chain financial activities.

DeFi Protocols. DeFi protocols leverage smart contracts to offer permissionless financial services such as lending (e.g., Aave [1]) and asset management. Two key primitives underpin these systems: (1) *Token swaps*, typically implemented by AMMs, enable decentralized trading against liquidity pools; (2) *Flash loans* (e.g., Uniswap [60]) allow users to borrow assets without collateral within a single transaction. Composability (*i.e.*, the integration of these primitives) enables complex financial workflows but also amplifies risk.

DeFi Attacks. The atomic and composable nature of DeFi can be exploited for complex attacks. In 2025, the incidents caused >\$2.9B in losses [50]. Common attack vectors include reentrancy, price manipulation, and flash loan attacks. The attacker typically deploy *adversary contracts* to orchestrate the exploit, coordinating interactions with victim protocols and auxiliary services (e.g., DEXs), and our preliminary analysis of 300+ incidents shows that **>90% of attacks involve adversary contracts**.

Example: WebkeyDAO Attack [65]. Figure 1a shows a partial invocation flow of a real-world WebkeyDAO attack [65]. The attacker (0x3026C46...) deployed an adversary contract (DODOFlashloan) and triggered the exploit via the wheeaappP function. During execution, the adversary contract coordinated interactions with the other contracts (e.g., wkeyDAO) and executed callbacks (Line 6), performing buy/sell operations (Lines 9–12) that ultimately transferred assets, yielding ~\$737,000 in profit.

2.2 Existing Techniques and Limitations

To analyze on-chain attacks, several techniques have been proposed, ranging from low-level execution tracing to static decompilation and manual PoC construction. While each approach provides valuable insights, they face distinct limitations in scalability, automation, and semantic precision.

Tracing. Tracing is a fundamental technique for analyzing on-chain transactions, as it records all executed low-level instructions (*i.e.*, execution traces) across contracts. In the running example, the full instruction-level trace contains >132 million EVM instructions. To improve usability, Ethereum clients (e.g., Geth [15]) provide call tracers that retain only call-related events, reducing the trace to ~5,000 instructions. Commercial tools [39, 42, 58] further enrich call traces with external knowledge (e.g., ABIs, and name tags) to facilitate understanding (e.g., the contract name DLP in Line 2).

However, even call-level traces remain difficult to analyze in practice. They still involve complex cross-contract interactions (15 contracts and ~5K calls in our example) and lack explicit data-flow semantics (e.g., the value 2.3e11 in Line 11 corresponds to the return value of a preceding call). Moreover, folded execution structures (e.g., loops in Line 12) can substantially inflate trace length, obscuring the underlying attack logic. Our empirical study of >300 incidents confirms this complexity: attack traces contain on average >770K executed instructions, involve 17 contracts, and include >1,700 cross-contract calls. As a result, tracing alone provides rich but low-level evidence, requiring labor-intensive manual inspection to reconstruct the underlying exploit logic.

```

[Attacker] 0x3026C464d3Bd6Ef0CeD0D49e80f171b58176Ce32
1 CALL [DODOFlashloan].wheeaappP...
2 STATICCALL [DLP].._BASE_TOKEN_()
3 CALL [DLP].flashLoan(, assetTo=[DODOFlashloan], ...)
4 DELEGATECALL [DVM].flashLoan...
5 CALL [USDT].transfer([DODOFlashloan], ...)
6 CALL [DODOFlashloan].DVMFlashLoanCall...
7 CALL [USDT].approve...
8 CALL [wkeyDAO].approve...
9 CALL [0xd511_Proxy].buy()
10 STATICCALL [wkeyDAO].balanceOf...
11 CALL [PancakeSwap].swapExactTokens(2.3e11,...)
12 ... Repeating 67x buy-balanceOf-swapExactTokens ...
13 CALL [USDT].transfer([Attacker], ...)

1 function wheeaappP(...) public payable {
2   require(4 + (msg.data.length - 4) - 4 >= 192);
3   ...
4   MEM[32 + MEM[64]] = varg0;
5   MEM[32 + MEM[64] + 32] = varg2;
6   MEM[32 + MEM[64] + 64] = varg1;
7   v0, /* address */ v1 = varg0.._BASE_TOKEN_();
8   require(bool(v0), 0, RETURNDATASIZE());
9   require(MEM[64]+RETURNDATASIZE()-MEM[64]>=32);
10  if (address(v1) - varg2) {
11    v2 = new uint256[](96);
12    MCOPY(v2.data, MEM[64] + 32, 96);
13    MEM[v2.data + 96] = 0;
14    require(bool(varg0.code.size));
15    v3 = varg0.flashLoan(0, varg1, address(this), v2);
16    require(bool(v3), 0, RETURNDATASIZE());
17  } else {
18    ...} // omit another branch
19 }

1 static call [0x107F3B...]._BASE_TOKEN_()
2 if address(ext_call.return_data[0]) != 0x55d398...:
3   call [0x107F3B...].flashLoan(uint256 baseAmount,
4                                uint256 quoteAmount, address assetTo, bytes
5                                data) with:
6   args 0, 1200 * 10^18, address(this), abi.encode
7   (0x107F3B..., 0x55d398..., 1200 * 10^18)

(c) Lifted wheeaappP Pseudocode from Trace.
1 function wheeaappP() public {
2   address base = IDVM(dvm).._BASE_TOKEN_();
3   if (base != USDT) {
4     bytes memory data = abi.encode(dvm, USDT, 1200
5     ether);
5   IDVM(dvm).flashLoan(0, 1200 ether, address(this),
6   , data);
7 }

(d) Lifted wheeaappP Code in Our PoC.

```

Figure 1: Running example from the WebkeyDAO attack [65]. (a) involves >15 contracts, ~5K cross-contract calls, and 132 million low-level EVM instructions. (b) shows decompiled wheeaappP code produced by SOTA decompiler [22, 23], exposing numerous unoptimized memory-related operations (red color). (c) presents our trace-driven lifted pseudocode. (d) shows the lifted wheeaappP function in the PoC, which captures the core attack logic with explicit semantics (e.g., `address(this)`), lifted unknown call targets `dvm` and parameters `data`, while eliminating low-level memory artifacts.

EVM Decompilation. Since most on-chain contracts (~99%) are not open-sourced (*i.e.*, verified) [16], another common approach is to decompile attacker bytecode into human-readable code [8, 22, 23, 28, 35, 41, 54].

Despite these efforts, decompiled outputs are often insufficient for precise recovery of attack logic. First, call parameters and intermediate values may remain unresolved [77] (e.g., variable `v2` in Line 15), because static decompilers are largely path-insensitive (Lines 4–15) and conservatively merge multiple execution paths. Consequently, memory states specific to feasible attack paths cannot be reconstructed precisely. More fundamentally, even along a single feasible path, EVM-specific dynamics (*e.g.*, unknown loop bounds, runtime-dependent parameters, and call return values) hinder accurate memory modeling. We model EVM memory as

$$D_M : (range, len) \mapsto e,$$

where both memory regions (`range, len`) and semantic expressions `e` may remain symbolic, causing frequent aliasing and preventing reliable elimination of low-level artifacts (*e.g.*, `MEM`). Finally, because static decompilation over-approximates all possible branches, isolating the core attack logic (*e.g.*, Line 18) becomes difficult, especially when adversaries deliberately obfuscate control flow or distribute logic across contracts. Our evaluation confirms that even advanced decompilers retain substantial low-level artifacts, limiting their usefulness for exploit reconstruction.

PoC Generation. Constructing proof-of-concept (PoC) exploits is a widely adopted strategy for validating and understanding attacks. Frameworks such as FOUNDRY [19] support this process by enabling historical state replay and account impersonation. Community efforts like DeFiHACKLABS [10] have curated over 670 PoCs, providing valuable reference implementations. Nevertheless, PoC construction remains largely manual and labor-intensive. Our measurement shows that ~50% of real-world incidents still lack corresponding PoCs (see § 5), creating a significant coverage gap.

In summary, while tracing, decompilation, and manual PoC construction have become indispensable for analyzing on-chain attacks,

their limitations in scalability, automation, and semantic accuracy prevent comprehensive coverage of real-world exploits. This motivates our work: *we propose to harness the code generation capabilities of LLMs to automate PoC construction, bridging the gap between low-level execution traces and high-level, reproducible exploit logic.*

3 Overview

3.1 Problem Formalization

Goal. We aim to automatically synthesize proof-of-concept (PoC) at the *source-code level* from observed attack transactions. Given the execution traces $T = \langle i_1, \dots, i_n \rangle$ during an attack, TRACExp lifts these low-level instructions i into a source-level exploit program P that encodes the adversarial execution logic. The synthesized program P is considered correct if, when executed under a compatible on-chain state, it reproduces the execution behavior and unintended asset (*e.g.*, USDT in our example) transfers observed in the original attack. By integrating with the industry-standard FOUNDRY framework [10, 19], TRACExp produces executable PoCs that can be readily used for downstream security analysis, such as root-cause investigation, and countermeasures deployment to mitigate financial losses.

Note that TRACExp does not aim to detect attacks or identify root-cause vulnerabilities, instead, it focuses on synthesizing PoC from observed attack transactions. As such, our work is orthogonal to prior efforts on attack detection and root-cause analysis (see § 7). Moreover, unlike existing approaches that generate PoC by analyzing vulnerable contract code [20, 27, 66, 69], TRACExp does not rely on access to contract source code, making it complementary to code-centric vulnerability analysis techniques.

Threat Model and Scope. We consider an adversary whose primary motivation is financial gain. Upon discovering a vulnerability, the adversary may initiate exploits through various vectors: direct interaction with victim contracts, deploying attack contracts, or executing complex multi-step transaction sequences. We make no assumptions about the structure, transparency (*i.e.*, open or closed source), or obfuscation level of the deployed attack contracts.

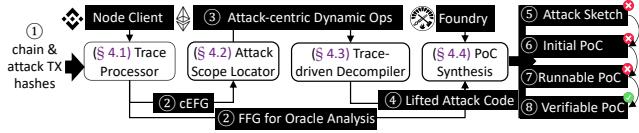


Figure 2: Workflow of TRACExp.

This work focuses on generating PoC code from attack transactions, including both launched transactions and pending transactions detected in the mempool. Consistent with empirical observations that ~95% of DeFi exploits utilize an intermediary attack contract [44, 64, 74, 77], we assume by default that the transaction is initiated against an adversary-controlled contract (§ 2.1). But TRACExp is flexible: if an attack interacts directly with a victim contract, our synthesizer can be configured to generate direct calls to reproduce the exploit. Currently, TRACExp targets EVM-based blockchains (e.g., Ethereum, BNB Smart Chain), leveraging the maturity of EVM analysis tools [22, 23, 41, 54] and the high density of real-world incidents in these ecosystems [10].

3.2 Challenges and Solutions

Synthesizing verifiable PoC from attack transactions poses challenges along two dimensions: recovering attack logic from low-level execution traces, and generating runnable, verifiable source code.

(1) Extracting Attack Logic from Low-Level Traces. Execution traces (e.g., via `debug_traceTransactions` [15]) record every low-level operation during a transaction, but this granularity introduces substantial noise. First, attack transactions involve complex interactions among adversary contracts, victim protocols, and peripheral components (e.g., tokens), making it non-trivial to isolate the attack-relevant sub-trace (C1). Second, traces capture only a single concrete execution path: high-level control structures such as loops are fully unrolled, and concrete values lack symbolic semantics (e.g., `address(this)`). These properties destroy the structural and semantic abstractions required for program recovery. Existing EVM decompilers are designed for static bytecode analysis and cannot directly lift such dynamic, unrolled traces, while inherent challenges in precise memory modeling further hinder lifting (C2).

(2) Synthesizing Verifiable PoC Code via LLMs. Although LLMs demonstrate strong general-purpose code generation capabilities [31, 33], applying them to exploit reproduction introduces additional challenges. First, exploit strategies vary significantly across DeFi protocols, requiring domain-specific knowledge and contextual reasoning that generic LLM prompts cannot easily capture (C3). Second, LLMs are prone to hallucinations, resulting in syntactic errors or semantic deviations from the original exploit. Ensuring that generated PoC code is not only plausible but also runnable and verifiable is therefore a central challenge (C4).

TRACExp Overview. To address these challenges, we design TRACExp, a framework that automatically synthesizes verifiable PoC from attack transactions. As shown in Figure 2, TRACExp first constructs a contract-centric execution representation to isolate attack-relevant behaviors (①-③), addressing trace noise and relevance (C1). Then, it applies a trace-driven decompilation approach that fuses concrete execution evidence with symbolic lifting to recover

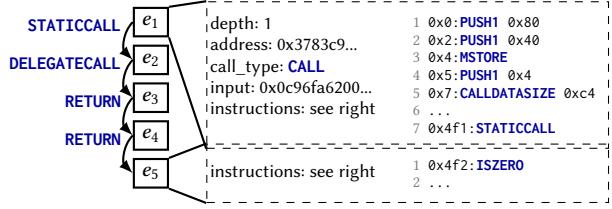


Figure 3: The cEFG of WebkeyDAO attack.

high-level attack logic (④) despite unrolled control flow and dynamic semantics (C2). To guide LLM-based synthesis, TRACExp extracts attack-specific semantic context (e.g., contract names and fund flows) to draft structured PoC sketches (⑤), mitigating domain diversity (C3). Finally, TRACExp employs an exploit-aware refinement loop that prioritizes semantic exploitability over input-output equivalence (C4). It first validates whether the generated PoC (⑥-⑧) reproduces unintended asset transfers, and then differentially refines execution behaviors to ensure verifiability and correctness.

4 Detailed Design

This section presents the components of TRACExp, detailing the transition from raw transaction traces to verifiable PoC code.

4.1 Trace Processor

This component reconstructs full execution traces from raw transaction hashes and transforms low-level instruction streams into a structured representation suitable for semantic analysis.

Contract-Centric Execution Flow Graph (cEFG). We introduce a *contract-centric* Execution Flow Graph (cEFG), extending prior EFG designs [79] to capture cross-contract exploit logic compactly. Each node is a 5-tuple $\langle \text{depth}, \text{address}, \text{call_type}, \text{input}, \text{ins} \rangle$, representing the call context and sequential instructions, nodes sharing identical context are aggregated to reduce redundancy. Here, *depth* is the call nesting level, *address* identifies the executing contract, *call_type* indicates the invocation opcode (e.g., CALL), *input* captures calldata, and *ins* lists instructions within that context. To optimize the graph for exploit analysis, we exclude instructions in the node whose *call_type* is STATICCALL, as they do not modify contract states and are typically auxiliary to the core attack logic. This structure preserves call-and-callback relationships and is particularly effective for multi-contract exploits.

Example: cEFG of WebkeyDAO attack. Figure 3 shows some nodes of a cEFG for a WebkeyDAO attack transaction. Execution starts at e_1 , the root context, and creates a new node e_2 when a STATICCALL triggers a context switch. Returns from nested calls (e_4-e_5) to the previous context. e_1 and e_5 share the same context, and their instructions can be aggregated together for the wheeaappP invocation.

Selective Argument Recording. Due to the dynamic semantics of the EVM, most notably data-dependent control dispatch (e.g., calldata-computed jump targets) and memory operations whose behaviors depend on runtime state (e.g., `returnData` produced by external calls), purely static analysis is insufficient to precisely reason about control-flow [75], fund-flow [48] and memory behavior (e.g., unoptimized memory in Figure 1b). Therefore, in addition to recording all executed instructions, we also record selected concrete

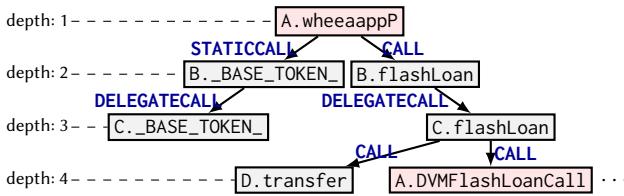


Figure 4: Call Graph. Adversary logic is in the red block.

runtime values to assist subsequent optimization and analysis. Consequently, TRACEXP selectively records concrete values for 5 critical instruction categories required by the *dual-decompiler* (§ 4.3):

- **Constants:** Immediate values from PUSH operations (e.g., Line 1).
- **Control Flow:** The conditional operands of JUMPI to facilitate control-flow graph reconstruction.
- **External Context:** Instructions interacting with external contract states or code (*i.e.*, EXTCODECOPY, CODECOPY, CODESIZE, and EXTCODESIZE), and the arguments of call-family instructions and their return values (*i.e.*, CALL, CALLCODE, DELEGATECALL, STATICCALL, RETURNDATASIZE, and RETURNDATACOPY).
- **Internal State:** Instructions accessing the contract’s persistent or transient storage, specifically SLOAD and TLOAD.
- **Input Data:** Calldata-related instructions (*i.e.*, CALLDATACOPY, CALLDATASIZE, and CALLDATALOAD) to track data flow from the transaction initiator.

Trace Implementation. We implement a *non-intrusive* customized tracer [18] compatible with standard Ethereum clients (e.g., Geth [15]). By hooking into the step function of the EVM, the tracer monitors execution in real-time. Upon detecting a context-switching opcode, the tracer dynamically instantiates a new cEFG node and begins capturing the execution metadata for the subsequent context. This approach enables high-fidelity dynamic analysis without requiring modifications to the underlying blockchain client source code.

4.2 Attack Scope Localization

To enable precise and scalable PoC synthesis, TRACEXP first localizes the *attack scope* within a transaction trace. Although a single attack transaction may involve many contracts, the exploit logic is typically confined to adversary-controlled contracts and a small number of their function invocations. Concretely, we define the attack scope as the set of adversary-related contract functions and their executed instructions that collectively realize the exploit.

Call Graph. TRACEXP constructs a hierarchical call graph (*CG*) by traversing the cEFG (§ 4.1). As shown in Figure 4, each node in the *CG* represents a functional execution context, uniquely identified by the tuple $\langle \text{address}, \text{depth}, \text{input} \rangle$. This representation enables function-level reasoning even when contract source code or precise ABI information is unavailable. Edges between nodes represent the *call type*, preserving the inter-contract invocation hierarchy.

Scope Identification Logic. Rather than relying on protocol-specific heuristics, TRACEXP localizes the attack scope based on execution semantics and adversarial control. Specifically, exploit logic resides in execution contexts that are directly controlled by

the attacker or derived from such contexts during execution. Accordingly, TRACEXP performs a conservative traversal of the call graph, starting from the initial recipient contract A_0 of the attack transaction. The attack scope is expanded based on the following criteria:

- **Direct Invocations:** All functions of A_0 executed during the transaction.
- **Dynamic Instantiation:** Contracts dynamically deployed by A_0 or its descendants via CREATE or CREATE2.
- **Context Delegation:** Functions executed via DELEGATECALL from any identified attack contract, as these executed external code is within the malicious contract’s context.

This localization stage substantially reduces the analysis scope. Compared to analyzing the full trace, localization decreases the number of executed instructions, inter-contract calls, and involved contracts by factors of 21 \times , 9 \times , and 13 \times , respectively. TRACEXP achieves a localization recall of **99.17%**, with false negatives observed in only three attack incidents. In these cases, TRACEXP generates exploitable PoCs by directly invoking the corresponding attack functions rather than reconstructing the attack contract code, which does not affect exploit reproduction correctness.

Instruction Extraction. During call graph construction, each identified functional context is mapped back to its corresponding node in the cEFG. Because each cEFG node is uniquely bound to an *address*, *input*, and *depth*, TRACEXP incrementally aggregates all instruction blocks belonging to each attack-relevant function. This process preserves internal state transitions and external call arguments, yielding complete instruction streams for subsequent trace-driven decompilation and PoC synthesis.

Example: CG of WebkeyDAO attack. Figure 4 shows the call graph built from the cEFG. The root node $A.wheeaappP$ represents the initial invocation of the adversary contract A by the attacker. After this stage, we further locate the $DVMFlashLocanCall$ function.

4.3 Trace-driven Decompiler

TRACEXP addresses a fundamental gap between symbolic decompilation and execution tracing by introducing a *trace-driven decompiler* that reconstructs high-level exploit semantics from a *single* observed execution. While symbolic decompilers recover structured code, they often blur concrete attack logic due to conservative abstractions, whereas execution traces precisely capture runtime behavior yet remain semantically opaque. Building on this observation, TRACEXP fuses symbolic structure with trace-observed concrete values to deterministically lift an executed exploit into concise, attack-focused pseudocode. At the core of this design are two tightly coupled components: (i) a *dual decompiler* (§ 4.3.1) that integrates concrete values into symbolic lifting, and (ii) a *trace compression* (§ 4.3.2) that recovers iterative semantics from unrolled traces.

4.3.1 Dual Decompiler. We propose a *dual decompiler*, a trace-driven semantic lifting framework that reconstructs high-level pseudocode from an already-executed trace with symbolic semantics. Unlike static decompilation, which lacks concrete runtime information, and concolic or symbolic execution [24, 47], which reason

about multiple feasible paths via constraint solving, the dual decompiler deterministically lifts a single observed execution into a semantically meaningful representation.

The dual decompiler augments an existing symbolic decompiler with concrete runtime values recorded during tracing (§ 4.1). The design is engine-agnostic and requires no modification to the underlying lifting logic. Conceptually, the dual decompiler operates through two synchronous processes: (i) *symbolic lifting*, which recovers high-level structure using standard decompilation techniques, and (ii) *concrete propagation*, which injects trace-observed values to refine and optimize the symbolic representation.

Concrete Integration via Binding. The core of the dual decompiler is *concrete integration*: symbolic expressions are explicitly anchored to concrete runtime values. To realize this, we introduce a *concrete-bind hook*, which extends the symbolic domain to a paired domain $\langle e, c \rangle$, where each symbolic expression e is associated with its concrete value c observed at the same execution point. This binding enables concrete values to be propagated through the symbolic representation, while symbolic expressions retain semantic meaning for otherwise opaque constants.

Concrete integration immediately yields *deterministic control-flow recovery*. Since branch conditions are bound to concrete values at each execution point, their concrete values are explicitly known during decompilation. As a result, the decompiler follows the *unique* branch, rather than speculating over alternative successors or exploring multiple paths. This trace-guided control-flow reconstruction ensures that the recovered control-flow graph precisely matches the observed exploit execution.

Precise Memory and Local Variables Modeling. Concrete integration further enables precise and efficient modeling of memory and local variables. Building on the memory abstraction introduced in § 2, we re-model the memory into

$$D_M^c : (range^c, len^c) \mapsto \langle e, c \rangle,$$

where the memory region $(range^c, len^c)$ is represented using concrete values. This design choice simplifies memory access semantics and enables deterministic resolution of *reads* and *writes* during decompilation. For memory writes, the accessed memory range (r, ℓ) is concretely determined by the execution. Concrete ranges may partially overlap; however, using deterministic segmentation of overlapping concrete segments (as implemented in existing static analysis tools, e.g., the `split_*` functions in PANORAMIX [41]), all memory accesses can be mapped to non-overlapping segments for reads and writes. By operating on these concrete segments, the dual decompiler deterministically resolves memory accesses without introducing aliasing ambiguities, and without requiring symbolic pointer reasoning or constraint solving.

Local variables are modeled analogously using a variable map

$$D_V^c : var \mapsto \langle e, c \rangle,$$

which is updated and queried deterministically.

Pseudocode Generation. With concrete control-flow and precise memory and local-state modeling in place, the dual decompiler performs trace-driven semantic lifting to reconstruct high-level pseudocode. Algorithm 1 summarizes this process. The algorithm

Algorithm 1 Trace-driven Dual Decompiler

```

DUALDECOMPILER( $P, D_V^c, D_M^c$ )
1:  $P' \leftarrow []$                                  $\triangleright$  output: high-level pseudocode with concrete bindings
2: for each statement  $s \in P$  do
3:   if  $s.type = \text{setmem}$  then
4:      $(r, \ell) \leftarrow \text{CONCRETERANGE}(s)$            $\triangleright$  concrete memory range
5:      $(e, c) \leftarrow \text{LIFT}(s.value, D_V^c, D_M^c)$      $\triangleright$  lift value with concrete-bind
6:      $D_M^c[(r, \ell)] \leftarrow \langle e, c \rangle$             $\triangleright$  update memory map deterministically
7:   else if  $s.type = \text{setvar}$  then
8:      $(e, c) \leftarrow \text{LIFT}(s.value, D_V^c, D_M^c)$ 
9:      $D_V^c[s.var] \leftarrow \langle e, c \rangle$                  $\triangleright$  update local variable map
10:  else if  $s.type = \text{if}$  then
11:     $(e_{cond}, \_) \leftarrow \text{LIFT}(s.cond, D_V^c, D_M^c)$      $\triangleright$  deterministic branch
12:     $B \leftarrow \text{DUALDECOMPILER}(s.branch, D_V^c, D_M^c)$ 
13:     $P'.append((\text{if}, e_{cond}, B))$        $\triangleright$  use symbolic condition semantic  $e_{cond}$ 
14:  else                                      $\triangleright$  handle other statement types: calls, arithmetic, etc.
15:     $(e, \_) \leftarrow \text{LIFT}(s, D_V^c, D_M^c)$ 
16:     $P'.append(e)$ 
17: return  $P'$                                  $\triangleright$  final pseudocode

```

sequentially interprets trace statements, lifts values via the concrete-bind hook, updates D_M^c and D_V^c deterministically, and emits symbolic expressions annotated with concrete bindings.

Overall, the dual decompiler reconstructs semantically meaningful pseudocode from a single, concrete execution trace under symbolic structure. It neither generalizes beyond observed behavior nor speculates on unexecuted paths, distinguishing it fundamentally from symbolic or concolic execution. Figure 1c shows the optimized decompiled pseudocode. Compared with existing outputs (see Figure 1b), temporary variables and memory references (Mem) are eliminated, significantly enhancing code readability.

4.3.2 Trace Compression. Execution traces fully unroll loops, producing linear sequences that can obscure high-level logic and inflate token consumption for downstream LLM-based synthesis (evaluated in § 5.4). To mitigate this, we introduce a lightweight *trace compression* module to summarize repetitive structures. Unlike traditional static analysis that recovers loops from a CFG [38], this module leverages the *deterministic iteration counts* inherent in the trace to accurately identify and parameterize loops.

Deterministic Loop Identification. Let a trace be represented as a sequence of concrete program counters $\mathcal{T} = [\text{PC}_1, \dots, \text{PC}_n]$. Loop identification aims to locate the *loop entry* e_ℓ , *exit* x_ℓ , the exact number of *iterations* k_ℓ , and the corresponding *loop bodies* $\mathcal{B}_\ell = [B_1, \dots, B_{k_\ell}]$. We detect consecutively repeated PC subsequences within \mathcal{T} , which deterministically define e_ℓ , x_ℓ , and k_ℓ , providing concrete loop bodies B_i for subsequent summarization. Formally, a repeated subsequence $S = [\text{PC}_p, \dots, \text{PC}_q]$ is recognized as a loop body if $\exists m \geq 2$ such that S occurs consecutively m times in \mathcal{T} .

Parameterized Loop Summarization. Once loop bodies \mathcal{B}_ℓ are identified, repeated statements are abstracted into a single *parameterized template* $\hat{\mathcal{B}}_\ell$. We perform a pairwise diff between adjacent loop bodies to classify operations as either invariant or variant. Specifically, for each statement s in the loop body:

- **Invariant:** If s is identical across all iterations, it is retained directly in the loop template $\hat{\mathcal{B}}_\ell$ and executed once per iteration, controlled by a loop index variable v_ℓ (e.g., i).

```

1 pragma solidity ^0.8.10;
2 import "forge-std/Test.sol"; // foundry test framework
3 import "../interface.sol"; // import some implemented interfaces
4 <Constant Addresses> // e.g., address constant attacker=0x...
5 contract ContractTest is Test {
6   function setUp() public {
7     vm.createSelectFork("chain", {blocknumber: -1}); // fork chain states
8     <Balances Assumptions> // e.g., deal(attacker, 1 ether);
9   }
10  function testPoC() public {
11    <Pre-Attack Balances> // e.g., emit log_named_uint("before attack", ...);
12    vm.startPrank(attacker); // set the msg.sender as attacker
13    AttackerC attC = new AttackerC(); // create attacker contract
14    <Attack Calls> // e.g., attC.attack{value: ...}(...);
15    vm.stopPrank();
16    <Post-Attack Balances> // e.g., emit log_named_uint("after attack", ...);
17  }
18 }
19 contract AttackerC { // the core attack logic implementation
20   <Attack Functions> {<Attack Logic>} // e.g., function attack(...);
21   <Other Functions> // e.g., callback functions | (...) Provided Attack Context |
22 }
23 <Other Contracts> // e.g., helper contracts | (...) Instrumented Oracle Logs |
24 <Involved ABIs> | (...) PLACEHOLDER for LLMs |

```

Figure 5: PoC Sketch Template in DEFICHLABS [10] Style.

- **Deterministic Variation:** If s varies according to a predictable function of the loop index (e.g., sequential memory addresses, array indices, or arithmetic over v_ℓ), we represent s as a deterministic function $f_s(v_\ell)$ within the loop template.
- **Complex Divergence:** If s exhibits irregular or non-monotonic changes, we introduce a temporary symbolic variable t_s (e.g., arr1) and express the statement in the template as $s[t_s, v_\ell]$ (e.g., arr1[i]), preserving the data flow across iterations.

Formally, the parameterized loop template is

$$\hat{B}_\ell = \{s \mapsto \text{Invariant} \cup f_s(v_\ell) \cup s[t_s, v_\ell] \mid s \in B_1\},$$

where each $s \in B_1$ corresponds to the first statement in \mathcal{B}_ℓ .

This trace-driven abstraction produces a compact representation that preserves the semantic behavior of the original execution.

4.4 PoC Synthesiser

The final stage of TRACEXP leverages LLMs to transform the optimized pseudocode into a verifiable source-level PoC. To ensure reliability, we decouple the synthesis process into *Sketch Generation* (§ 4.4.1) and *Iterative Refinement* (§ 4.4.2).

4.4.1 Sketch Synthesis. Directly prompting an LLM to generate exploit code is often infeasible due to: (1) *Safety Alignment*: LLMs' built-in mechanisms may refuse to produce malicious payloads¹; and (2) *Framework Dependencies*: PoC in modern testing framework (e.g., FOUNDRY [19]) require precise environmental configuration and metadata (e.g., block forks), which are difficult for LLMs to infer implicitly. Rather than treating the LLM as a standalone code generator, TRACEXP adopts a *sketch-based synthesis* strategy that decouples *analysis* from *code completion*.

Sketch Components. At a high level, TRACEXP first derives a structured PoC sketch (see Figure 5) from dynamic execution evidence and then constrains the LLM to complete only well-defined placeholders, which both reduces reliance on unconstrained LLM

¹Although some jailbreak strategies [12] were proposed, this is not our focus.

generation and ensures compatibility with real-world exploit testing environments. The sketch comprises three main components:

- **Rule-derived Attack Context:** Pre-configured execution environment data tailored to the specific attack (e.g., Line 7).
- **Instrumented Oracle Logs:** Logic-based logs automatically embedded into the harness for exploit verification (i.e., Lines 11, 16).
- **Functional Placeholders:** Structural stubs that the LLM populates using the decompiled pseudocode (e.g., Line 21).

Specifically, the Rule-derived Attack Context encompasses: (i) *Environmental Configuration* (Lines 6–9), which defines the blockchain network (e.g., Ethereum, BSC), the specific fork block number, and the minimum funding requirements derived from fund-flow analysis; (ii) *Metadata and Interfaces*, including target contract addresses, human-readable aliases (Line 4), and essential ABI specifications (Line 24); and (iii) *Invocation Sequence* (Line 14), which provides a high-level roadmap of the function calls the PoC must execute.

Fund Flow and Oracle Extraction. Correctness of a synthesized PoC is defined by whether it reproduces the *semantic outcome* of the original exploit, rather than merely executing without errors. In DeFi attacks, such outcomes manifest as observable *economic revenue* (i.e., asset redistribution) across accounts, which can be precisely recovered from execution traces. TRACEXP models the fund flow of a transaction as net balance changes over all involved assets. Formally, for an account a and an asset t (including ether and ERC20 tokens), we define

$$\Delta(a, t) = \text{balance}_{\text{post}}(a, t) - \text{balance}_{\text{pre}}(a, t).$$

Non-zero $\Delta(a, t)$ values collectively characterize the economic effect of the transaction. TRACEXP first evaluates the net profit of the sender (i.e., attacker) and the attack contract. If no significant gain is detected, we iterate through all involved addresses to identify the entity with the maximum net profit as the presumed beneficiary.

To extract fund flow, TRACEXP monitors trace instructions that induce asset transfers. Token movements are identified via LOG-based events (i.e., Transfer, Deposit, Withdrawal), while native ether transfers are captured through value-carrying instructions (e.g., CALL, CREATE(2), and SELFDESTRUCT). The extracted fund flow is used to derive the minimum attacker balance (Line 8) and to synthesize balance-change assertions that serve as deterministic oracles for exploit validation (i.e., Lines 11, 16).

LLM-driven Completion. Finally, TRACEXP provides the LLM with the ordered, decompiled pseudocode (§ 4.3) and instructs it to fill the functional placeholders. To ensure semantic fidelity during DELEGATECALL operations, where logic is executed within the caller's state context, we explicitly prompt the LLM to encapsulate such logic within distinct contract structures. This ensures the generated code faithfully preserves the runtime behavior of the original exploit.

4.4.2 Exploit-Aware Refinement. LLM-generated PoC code may suffer from both syntactic errors and semantic deviations. To ensure verifiability, TRACEXP introduces an *exploit-aware iterative refinement loop* that differs fundamentally from prior program repair or differential testing approaches. Instead of enforcing strict input–output equivalence, our refinement prioritizes *semantic exploitability*, i.e., whether the synthesized PoC reproduces the unintended asset transfer observed in the original attack.

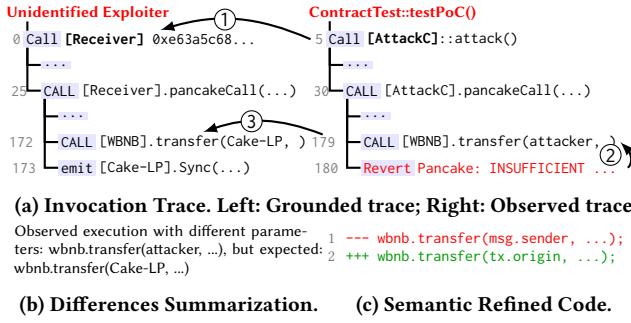


Figure 6: Illustration for Error-Directed Trace Alignment.

Syntax Refinement. We first eliminate syntactic inconsistencies using compiler-directed feedback. Upon compilation failure, TRACEXP extracts error messages from the FOUNDRY environment and constructs a focused repair prompt containing only PoC snapshot and the reported errors. Each refinement iteration is stateless, avoiding prompt accumulation and reducing token overhead. This process repeats until the PoC successfully compiles.

Semantic Refinement. A compilable PoC does not necessarily imply exploitability. TRACEXP therefore performs semantic refinement in a *two-tier, exploit-driven manner*: (1) *Profit-Oriented Validation*. We first evaluate whether executing the synthesized PoC yields the unintended asset transfer as the ground-truth attack. This profit oracle serves as a coarse-grained semantic criterion: if satisfied, the PoC is considered semantically correct, even if low-level execution details differ. (2) *Execution Alignment*. If the profit oracle fails, TRACEXP performs a targeted comparison between the execution trace produced by FOUNDRY and the original on-chain trace. Rather than enforcing full trace equivalence, we localize discrepancies that are likely to affect exploitability, such as mismatched call targets, incorrect addresses, or inconsistent argument values, and translate them into actionable feedback for the LLM.

Greedy, Error-Directed Trace Alignment. Direct trace comparison is challenging due to benign divergences caused by contract creation, address randomization, or call reordering. To robustly localize semantic mismatches, as illustrated in Figure 6, TRACEXP adopts a *greedy, error-directed* alignment strategy consisting of three steps: (1) we align contract addresses in the two traces according to their creation order to establish a coarse correspondence (e.g., AttackC and Receiver). (2) we greedily prioritize locations where FOUNDRY reports runtime errors (e.g., Line 180) and attempt repairs beginning at those error sites. (3) To infer the semantics that should have been executed at a problematic location, we perform a local, windowed matching: within a small neighborhood of the error, we compare call targets, argument patterns, and other call-site metadata to identify the call in the generated trace that most closely matches the expected operation. We then present the paired statements (the expected and the observed calls) to the LLM as focused context for semantic correction (e.g., Figure 6b). This combination of creation-order alignment, error-directed greedy repair, and localized window matching helps localize semantic differences and guides effective LLM-driven corrections (e.g., Figure 6c).

By combining syntax and semantic refinement, TRACEXP guarantees that the automatically generated PoC code is both compilable and semantically faithful, accurately reproducing the attack logic while remaining compatible with FOUNDRY.

5 Evaluation

In this section, we evaluate TRACEXP to demonstrate its effectiveness and efficiency in synthesizing executable PoC from low-level execution traces. As *no prior work* directly addresses this task, we compare TRACEXP with representative alternative approaches for exploit analysis and PoC generation, and conduct ablation studies to assess the contribution of each design component. Specifically, our experiments focus on the following research questions:

- **(RQ1)** How effective and efficient is TRACEXP in generating PoC for real-world attacks (§ 5.2–§ 5.3)?
- **(RQ2)** To what extent does each design component contribute to the overall effectiveness (§ 5.4)?
- **(RQ3)** How does TRACEXP compare against alternative methods for exploit reproduction and PoC generation (§ 5.5)?

5.1 Datasets and Experimental Setup

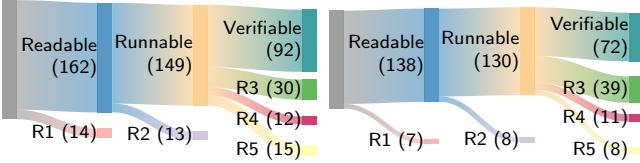
Datasets. To evaluate TRACEXP, we constructed TRACEXP-DS, a large-scale benchmark comprising 321 real-world attack incidents and their corresponding transaction hashes from the ETH and BSC networks. The dataset spans 20 months (January 2024 to July 2025). Compared to existing benchmarks, such as FORAY [66] (34 incidents) and A1 [20] (38 incidents), TRACEXP-DS is an order of magnitude larger. This scale is achieved because TRACEXP is agnostic to attack types, and the low computational cost of our pipeline enables high-throughput evaluation. The dataset is curated from two sources:

- **Exploits with Ground-Truth PoCs (*w-poc*):** We collected 178 incidents from DeFiHACKLABS [10]. We extracted “attack transaction hashes” primarily from analyst metadata, and used exploit reports as a fallback when unavailable. After filtering two cases (EGGX and LQDX) due to missing transaction records, we retained 176 incidents as our ground-truth set.

- **Exploits without Existing PoCs (*wo-poc*):** To evaluate the generality of TRACEXP on in-the-wild attacks, we aggregated over 300 alerts from prominent security firms, including CertiK [2], SlowMist [50], and TenArmor [57]. After deduplication against the *w-poc* set, we retained 145 unique incidents.

Implementation. TRACEXP utilizes PANORAMIX [41] as the core decompiler backend, adopting its output as the high-level Intermediate Representation (IR) for subsequent synthesis. Nevertheless, the modular design of TRACEXP ensures that it remains decompiler-agnostic and can be readily adapted to other decompilers [22, 23, 54]. We employ GPT-5 [21] as the primary LLM for PoC synthesis, given its strong performance on recent code reasoning and generation tasks [3, 29, 36]. We accessed the model via the OPENROUTER [40] unified API. In accordance with established best practices for software engineering tasks [62], we optimized the model parameters by setting `reasoning_effort` to *minimal* and `verbosity` to *low*.

Experimental Setup. To ensure a consistent and bounded evaluation, we imposed a 10-minute timeout for the trace processing and PoC sketch generation phases. For the iterative refinement loop,



(a) For 176 incidents with PoC. (b) For 145 incidents without PoC.

Figure 7: Results of TRACEXP on TRACEXP-DS. R1-R5 denote the reasons for unsuccessful cases: *Timeout*, *Syntax Repair Failure*, *Incomplete Context*, *Execution Failure*, and *Semantic Repair Failure*, respectively.

we set a maximum budget of 5 iterations for syntactic repair and 3 iterations for semantic refinement. During the semantic refinement stage, if a correction introduces new syntax errors, we permit up to 3 auxiliary syntactic repair attempts per semantic iteration. All experiments were conducted on an Ubuntu 22.04 server equipped with an Intel Xeon Gold 6252 CPU (2.10 GHz) and 251 GB of RAM.

5.2 Main Results for TRACEXP

To assess the effectiveness of TRACEXP, we evaluate the generated PoC along three dimensions:

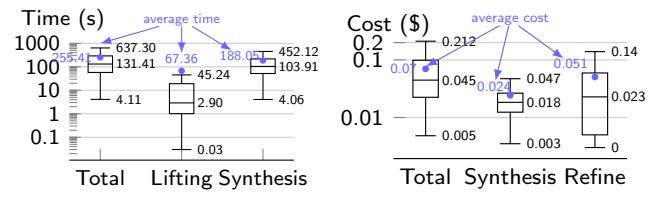
- *Readable*: TRACEXP successfully produces human-readable PoC source code from attack transactions within a fixed time budget;
- *Runnable*: the generated source code compiles successfully under the FOUNDRY framework;
- *Verifiable*: the compiled PoC not only runs but also reproduces the original attack by satisfying the predefined profit oracles.

While *verifiability* is our primary objective, *readable* and *Runnable* outputs are also practically valuable, as they provide structured templates that substantially reduce the manual effort required for exploit analysis and reproduction (see § 6).

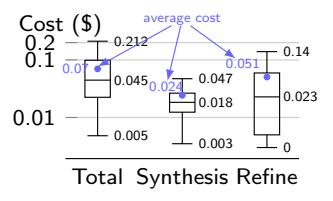
Overall Results. Figure 7 summarizes the evaluation results on TRACEXP-DS. Out of 321 exploit incidents, TRACEXP produces 300 (93.46%) *readable* PoC codes. Among them, 279 (93.00%) are *Runnable*, i.e., they compile successfully under the FOUNDRY framework without manual intervention. Finally, 164 of the runnable PoC codes (58.78%) are *verifiable*, successfully reproducing the original attack by satisfying the predefined profit oracles.

Real-World Impact. To assess the practical utility of TRACEXP, we anonymously submitted 33 generated PoC codes from the *wo-poc* subset to the DEFICHLABS [10] 2025 Summer Contest². These submissions accounted for 38% of all community contributions in August 2025. Notably, the number of PoC submissions generated by TRACEXP exceeded the 31-day output of five top human contributors by factors ranging from 1.32× to 33.00×. Measured on a per-attack-event basis, TRACEXP generated PoC submissions 16.50× to 33.00× more efficiently than the other 4 contributors. This result provides external evidence that TRACEXP can efficiently generate practically usable PoC artifacts at scale, both earning positive feedback from the community and also a **\$900 bounty**.

Time and Monetary Cost. Beyond effectiveness, we evaluate TRACEXP’s efficiency in terms of execution time and monetary



(a) Time Cost.



(b) Monetary Cost.

Figure 8: Time and Monetary Cost on TRACEXP-DS.

cost. Figure 8 presents Tukey-based boxplots of both metrics across TRACEXP-DS, decomposed by processing phase. Overall, TRACEXP completes PoC generation in 255.41s on average, with a mean monetary cost of \$0.07 per case.

To better understand runtime behavior, we decompose total execution time into **Trace Lifting**, which converts raw traces into structured PoC sketches, and **PoC Synthesis**, which involves iterative LLM interactions and FOUNDRY-based validation. During Trace Lifting, the median runtime is 2.90s, the mean is 67.36s, and the maximum is 600s due to timeout. Most cases (93.46%) complete within the limit, and 82.87% finish under 60s. For non-timeout cases (average runtime: 30s), instruction-level lifting and structural optimizations in the dual decompiler account for 22.86s on average, representing the majority of phase runtime. PoC Synthesis exhibits higher latency, with a median of 103.91s and mean 188.05s. The majority of runtime in this phase arises from network-dependent LLM inference, iteration, and on-chain state retrieval.

Monetary cost is decomposed into **Synthesis** (mean \$0.024) and **Refine** (\$0.051), totaling \$0.07 per case (Figure 8b). Synthesis incurs a largely stable cost, whereas Refine exhibits higher variability, reflecting differences in refinement effort, and the minimum Refine cost is \$0, indicating that some cases require no refinement. Further analysis of LLM token consumption and the number of refinement iterations reveals that most cases require a single syntactic correction, and some additionally require semantic refinements. Moreover, higher Refine costs are associated with increased token usage and repeated semantic corrections, especially for unresolved semantic inconsistencies could cause more monetary overhead in challenging cases.

LoC. Finally, beyond efficiency and effectiveness, we report a simple property of the generated PoCs as a complementary signal of understandability. On average, the PoCs contain 132 lines of code (median: 116; max: 588), suggesting that they remain human-scale rather than overly verbose artifacts.

5.3 Failure Analysis

To understand the limitations of TRACEXP, we analyze the failed cases across three dimensions: static analysis timeouts, unresolved syntactic issues, and semantic refinement failures (see Figure 7).

Timeout Analysis. 21 cases (6.54%) timed out, occurring during either the trace processor (§ 4.1) or the trace-driven decompiler (§ 4.3) stage. Despite occurring at different stages, all these cases share a common root cause: extreme execution complexity.

²The final leaderboard is available at <https://leaderboardhq.com/5kmsevrj>.

```

1 call 0x81917e....flashLoan(...,      1 bytes memory dpp2Data=
2 data=abi.encode(                  2 hex"14172fc41...
3   0x14172fc41...,                 3 hex"944490e6cb..." // error: Expected
4   0x944490e6cb...,                 even number of hex-nibbles.
5 )                                    4 ...
6 )                                    5 IDPPAdvanced.flashLoan(..., dpp2Data);

```

(a) prefix “0” are omitted in constant values (Lines 3-4). (b) Synthesized code but with compilation errors (Line 2).

Figure 9: Illustration of syntax errors in synthesized code (the transaction of this DualPool exploit is [90f374...](#)).

Specifically, one timeout occurs during trace processing, where the transaction³ cannot be fully traced due to extreme execution complexity. The remaining 20 timeout cases arise from excessive instruction volume in attack functions. For these cases, the longest functions identified at the locator stage (§ 4.2) contain on average ~262K instructions (median: 228K), which is 13× (mean) and 71× (median) larger than those in non-timeout cases. Further inspection reveals that such instruction explosion is typically driven by highly iterative execution patterns. In 10 cases, loops execute more than 1,000 iterations. For example, in the FIL314 attack⁴, execution involves 6,000 loop iterations, resulting in 2.776M instructions, with a single function contributing about 740K instructions.

Unresolved Syntax Errors. We find that the LLM can iteratively resolve most (93%) syntax errors (e.g., type mismatches or undeclared variables) through syntax refinement (§ 4.4.2), consistent with prior observations [54]. However, for those unrepairs syntax errors, >60% of them are attributed to parameter encoding issues. As shown in Figure 9a, decompilers often omit leading zeros from constants (Lines 3-4). This results in hexadecimal literals with an odd number of digits in the generated PoC (Figure 9b), which triggers a Solidity syntax error (Line 3). In the absence of original ABI information, the LLM lacks sufficient semantic context to determine the intended padding and thus fails to repair the malformed literal.

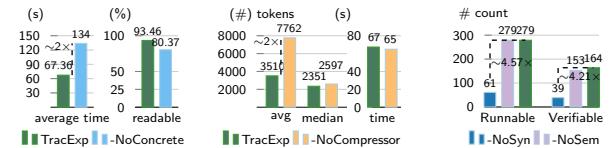
This issue stems from the closed-source nature of most attack contracts. Without access to ABIs, decompilers (e.g., PANORAMIX [41]) rely on heuristics to decode parameters, often producing raw constants with ambiguous semantics. The LLM subsequently propagates these constants verbatim into the generated PoC. A potential mitigation is to incorporate semantic-aware parameter analysis to help reconstruct the intended argument structure, thereby reducing encoding-related syntax failures.

Unresolved Semantic Errors. As shown in Figure 7, unresolved semantic errors arise from three distinct failure modes: (1) incomplete context (21.50%); (2) execution failures in FOUNDRY (7.17%); and (3) limitations in LLM-driven semantic refinement (7.17%).

(1) *Incomplete Context.* This failure mode occurs when intelligence sources omit preparatory transactions executed by the attacker prior to the exploit. In the absence of these state-setting steps, synthesized PoC are executed under invalid preconditions. We provide detailed examples, including missing allowance initialization in the MEEKOO contract, along with corresponding traces and PoC corrections. A potential mitigation is to track relevant

³The attack transaction is [0xc7927a...](#)

⁴The attack transaction is [0x9f23b1...](#), and the PoC is available at [FIL314_exp](#).



(a) **TRACEXP vs -NoCONCRETE** (b) **TRACEXP vs NoCOMPRESSOR** (c) **TRACEXP vs -NoSYN & -NoSEM**

Figure 10: Ablation studies of TRACEXP on TRACEXP-DS.

contract state variables during trace lifting and pre-initialize them in the FOUNDRY test harness prior to execution.

(2) *Foundry Execution Failures.* A subset of semantic failures is caused by instability in Remote Procedure Call (RPC) services. When the RPC provider fails to retrieve on-chain state or returns database errors, FOUNDRY is unable to simulate the transaction. Since the semantic refinement loop critically depends on successful execution feedback, such infrastructure-level failures interrupt the debugging process and prevent the synthesis of a verifiable PoC.

(3) *LLM Refinement Failures.* These failures stem from several complementary challenges. *First*, execution traces generated by FOUNDRY often diverge from the original exploit traces in terms of contract addresses and internal call structures, which hinders accurate semantic diffing and root-cause localization. *Second*, certain semantic defects reflect inherent limitations of current LLMs that cannot be resolved through simple prompting. For example, in the EXcommunity attack⁵, the LLM incorrectly interprets a custom send implementation as Solidity’s built-in Ether transfer primitive, leading to flawed logic. *Third*, deliberate obfuscation in attack contracts (e.g., computing values via sha3 over contract addresses) causes synthesized addresses to deviate from the originals, preventing exact reproduction of address-dependent behavior. These limitations point to the need for future research on robust state reconstruction, trace alignment, and semantics-aware code synthesis.

5.4 Ablation Studies

We conduct a stepwise ablation study to quantify the contribution of each core component in TRACExp. To balance evaluation depth with computational cost, we employ a two-stage assessment framework.

The *first* stage evaluates static analysis by measuring the quality of the intermediate PoC (*i.e.*, the PoC sketch) in code structure and redundancy. This stage includes two variants:

- **TRACEXP-NoCONCRETE**: omits the use of concrete values, evaluating the impact of concrete values on trace-analysis success rate (*i.e.*, whether or not sketch generation) and efficiency.
- **TRACEXP-NoCOMPRESSOR**: omits the compressor, evaluating if trace compression effectively summarizes loops, thereby reducing the token volume and improving human readability.

The *second* stage assesses PoC generation, measuring the *verifiable* ability and demonstrating the contribution of syntax and semantic refinement. This stage also contains two variants:

- **TRACEXP-NoSYN**: omits syntactic refinement.
- **TRACEXP-NoSEM**: omits semantic refinement.

⁵The attack transaction is [0x5446bf...](#), and the PoC is available at [EXcommunity_exp](#).

Figure 10 summarizes the ablation results, confirming that each design component contributes to the efficiency or effectiveness of PoC generation. **Figure 10a** shows that incorporating concrete-value analysis substantially improves efficiency: the average end-to-end runtime is **halved**, and the PoC generation success rate increases by $\sim 13\%$. **Figure 10b** demonstrates that the compressor algorithm effectively reduces the total token number provided to LLMs, decreasing the average input token volume by $\sim 55\%$. The median values are similar because we find that $<20\%$ of attacks contain heavy loops. However, for those loop-intensive cases the compressor can reduce the total token count dramatically (in extreme cases to about 1/100 of the original). The additional runtime overhead introduced by compression is negligible (<3 seconds on average).

For iterative refinement, **Figure 10c** reports the impact of syntactic and semantic refinement. Without syntax refinement, only a small fraction of generated PoCs are immediately runnable or verifiable ($\sim 20\%$ and 12.15% , respectively). Enabling syntax refinement increases the runnable and verifiable rates by factors of $4.57\times$ and $3.92\times$, respectively. Adding semantic refinement yields a further improvement: verifiability increases by an additional 3.43% .

5.5 Comparison with Alternative Approaches

In this section, we report the comparison results between TRACEXP and existing exploit generation approaches. To our best of our knowledge, we are the *first* to directly generate PoC code from raw transactions. Thus, we selected alternative approaches that generate exploits from *vulnerable contracts* and *attack contracts*:

- **A1 [20]**: It proposes an agentic system that transforms LLMs into smart contract exploit generators by equipping them with 6 domain-specific tools (e.g., a source-code fetcher) for analyzing *vulnerable* smart contracts. Here, we exclude ADVSCANNER [69] as it is not open-source and focuses exclusively on reentrancy vulnerabilities. We also exclude FORAY [66] since it requires substantial manual effort (*i.e.*, specifying attack targets, initial-state configurations, and additional function mappings) to generate exploits, which limits its scalability on general attacks.
- **DiSCo [54]**: It translates EVM bytecode into a natural language intermediate representation (IR) and then leverages LLMs to lift this IR into source code, which facilitates attack reproduction. As some of the lifted source codes are not directly executable, we manually adapted them into FOUNDRY-compatible code and manually validated whether they can reproduce the attacks. Here, we did not include other EVM decompilers (e.g., GIGAHOST [22] and PANORAMIX [41]) because their outputs are often complex *pseudocode* and typically omit logic in contract constructors [54].

Benchmarks. For a fair comparison, we evaluate TRACEXP on the dataset curated by A1 [20], which comprises 38 real-world exploits on Ethereum and BSC (2021–2025). This dataset includes 29 incidents from the VERITE benchmark [34] and 9 cases introduced by A1. Notably, while A1 excluded two cases due to unavailable source code, we include them to demonstrate TRACEXP’s independence from source-level information. We utilize the corresponding PoC codes from DEFIHACKLABS [10] as the ground truth for validation.

Overall Results. Overall, TRACEXP produced 30 verifiable PoCs (78.95%) with a monetary cost of \$0.0426 and a time cost of 142.75 seconds on average, representing relative improvements over the

baselines ranging from 10% to 68%. Across these incidents, TRACEXP generated PoC code for 95% of cases, and nearly 90% of the generated PoCs compiled successfully. These results indicate that TRACEXP substantially improves both the coverage of automatic PoC synthesis and the rate of producing compilable, executable exploits compared to prior approaches. Additionally, we observe that TRACEXP performs substantially better on this dataset than on TRACEXP-DS. Further investigation attributes this gap to differences in the complexity distribution of attack transactions across the two datasets. In particular, the incidents in this dataset require materially less analysis effort: their average runtime and monetary cost are only approximately 60% of those measured for TRACEXP-DS, which explains the improved performance.

Compared with A1 [20]. Since A1 is not open-source, we can only compare with its reported results. Beyond the final PoC generation result, we also compare the monetary cost. A1 relies on six different models, so for a fair comparison, we consider Gemini Pro with equivalent cost to GPT-5 (\$1.5/M for input, \$10/M for output). The average cost per case for A1 ranges from \$0.11 to \$0.19, while TRACEXP only incurs an average of \$0.0426, which is significantly lower. One contributing factor is that, for many cases (25 out of 38), the generated code already reproduces the attack effectively immediately after syntax repair, and 7 cases required no syntax refinement at all. In contrast, the inputs for A1 include long contract code, and it relies on interactions among multiple agents, which inevitably leads to additional token consumption.

Compared with DiSCo [54]. Although DiSCo can decompile EVM bytecode into source code, our experiments show that only 16 out of 38 attack contracts were successfully decompiled, while the others failed. This observation suggests that traditional decompilation tools may face efficiency issues when reversing complicated attack contracts. In contrast, TRACEXP analyzes only the transaction traces and also introduces concrete values, which may help alleviate this problem. Moreover, among the successfully decompiled contracts, manual inspection revealed that some code, particularly attack-related logic, was still missing. This finding aligns with DiSCo’s claim that LLMs may omit content they fail to understand.

6 Discussion

Data Leakage Concerns. Data leakage is a common concern when applying LLMs to code generation. We mitigate this risk through controlled inputs and empirical validation: the model only receives intermediate representations (*i.e.*, decompiled pseudocode and context-rich PoC drafts), and our evaluation includes many attacks without public PoC, with comparable performance across both settings. Following prior work [30, 54], we further assess potential leakage by ranking public PoCs by semantic similarity and manually inspecting 50 pairs of generated and public PoCs. We observe clear stylistic and structural differences and find no exact duplication, suggesting data leakage does not affect our conclusions. **Manual Efforts for Unverifiable PoCs.** Even some generated PoCs are not immediately verifiable, TRACEXP substantially reduces manual effort by producing complete PoCs that captures the correct exploit logic. For PoCs that fail verification, manual inspection confirms consistency with ground-truth attack logic, indicating implementation rather than reasoning errors. Further, we sample 10

PoCs with syntactic errors and 10 with semantic errors for manual repair. Syntactic issues are resolved within ~5 minutes on average, while semantic issues require ~20 minutes, except for FOUNDRY execution failures, all sampled cases can be repaired. Overall, TRACEXP shifts exploit reproduction from manual reverse engineering to lightweight debugging and refinement.

Attack Coverage. We evaluate TRACEXP across diverse attack patterns (§ 2) and observe consistent performance. Across all patterns, TRACEXP reproduces $49.17 \pm 4\%$ of attacks, suggesting that it does not rely on pattern-specific heuristics. Notably, TRACEXP remains effective for attacks *without* adversary contracts, successfully reproducing 64.29% of such events. This result indicates that the approach generalizes beyond adversary-contract-centric attack models, provided sufficient execution traces are available.

Applicability to Other LLMs. TRACEXP is designed to be LLM-agnostic and interacts with models through standard prompt-based interfaces. To validate this design, we further evaluate DeepSeek-R1 [25] and Gemini-2.5-Flash [6] on A1 datasets (§ 5.5). All these LLMs can generate PoCs, though the fraction of verifiable PoCs decreases by ~25% and 10%, respectively, reflecting differences in code generation and reasoning capabilities. Importantly, the pipeline remains functional across LLMs, and future improvements in base LLMs can directly improve end-to-end reproduction rates.

Limitations. This work has two main limitations: (1) when attack traces contain extremely long execution paths, static analysis may become slow or time out. This stems from our instruction-level lifting of concrete execution paths prior to loop abstraction. While CFG-first decompilation approaches [41] scale better, they may lose dynamic flow information critical for accurate reconstruction. A hybrid approach that selectively preserves dynamic information while leveraging CFG-based analysis could improve scalability without sacrificing fidelity. (2) PoC synthesis currently assumes that users can provide a complete set of attack transactions. Missing preparatory transactions or state updates may cause execution failures, and such state is sometimes difficult to recover in practice. Future work could mitigate this limitation by automatically reconstructing and initializing relevant contract state using inferred read/write dependencies from traces and on-chain state.

7 Related Work

In this section, we position TRACEXP in the context of execution trace analysis, automated exploit generation, bytecode analysis, and the emerging paradigm of LLM-based security analysis.

Execution Trace Analysis. Execution trace analysis has been widely studied for attack detection. Early systems such as SEREUM [46] and SODA [4] analyze traces using dynamic tainting and instruction-level pattern matching. To capture more complex attack logic, later approaches adopt graph-based representations, including TXSPEC-TOR [79], which constructs Execution Flow Graphs for rule-based reasoning, and CLUE [43], which integrates call, control, and data dependencies into an Execution Property Graph. Recent work further incorporates graph learning or pre-trained models to improve detection accuracy [53, 63, 70], with specialized systems targeting price manipulation [68, 71], flash loan exploits [5], and cross-chain attacks [67]. While TRACEXP shares the use of execution traces

to capture inter-contract dependencies, prior work primarily focuses on detection. In contrast, TRACEXP lifts concrete execution traces into human-readable code, bridging detection and verification through automated PoC synthesis.

Exploits Generation. To improve interpretability, several studies generate exploit transactions [5, 51] or contracts [20, 27, 66, 69]. SMARTEST [51] combines symbolic execution with LLMs to prioritize vulnerable transaction sequences, while FORAY [66] and AdvSCANNER [69] generate exploit code using static analysis and LLMs but require substantial manual refinement or target specific vulnerability classes. More recent agentic frameworks, such as A1 [20], CVE-GENIE [59], and SMARTPoC [3], enable autonomous exploit generation using execution-driven agents. Unlike these approaches, TRACEXP starts from confirmed attack transactions and directly reconstructs PoC code, enabling general exploit reproduction without relying on prior vulnerability detection or access to source code.

EVM Bytecode Analysis. Numerous EVM decompilers target pseudocode [17, 22, 23, 28, 35, 41, 83] or high-level source code [8, 54]. Pseudocode-oriented tools often preserve low-level semantics at the cost of readability, while source-level decompilers improve readability but may sacrifice semantic fidelity. Existing tools also struggle with obfuscation [48, 75], dynamic data types [74, 77], and typically aim to recover entire contracts rather than specific logic fragments. Rather than decompiling bytecode directly, TRACEXP reverses concrete execution traces, enabling concise and semantically accurate reconstruction of attack logic for PoC generation.

LLM-based Security Analysis. LLMs have been increasingly applied to smart contract security tasks, including logic vulnerability detection [7, 55, 76], DeFi price manipulation analysis [81], secure program partitioning [37], and similarity-based vulnerability identification [78]. Beyond smart contracts, LLMs have also been explored in broader cybersecurity settings, such as automated attack generation [72], enhanced decompilation [56, 73], repository-level auditing [26], and code review generation [32]. These efforts primarily leverage LLMs to improve vulnerability discovery, summarization, or reasoning over security-relevant artifacts. In contrast, TRACEXP employs LLMs for post-incident exploit reconstruction. Rather than detecting vulnerabilities or ranking alerts, we lift concrete execution into human-readable PoC, complementing prior LLM-based security analysis with a verification-oriented perspective.

8 Conclusion

We presented TRACEXP, the *first* automated framework for synthesizing PoC directly from on-chain attack executions. By bridging low-level execution traces with high-level exploit logic, TRACEXP enables systematic and reproducible understanding of real-world DeFi attacks without requiring source code or prior vulnerability knowledge. At its core, TRACEXP combines trace-driven reverse engineering with LLMs to distill concise, semantically faithful exploit logic, and validates synthesized PoCs based on exploitability-relevant semantics. Our evaluation on nearly 300 real-world attacks shows that TRACEXP can automatically generate over 50% verifiable PoCs for a large fraction of incidents. By lowering the barrier to post-attack analysis and enabling rapid attack reproduction at scale, TRACEXP supports more effective incident response and contributes to improving the security of the DeFi ecosystem.

References

- [1] aave 2025. aave. <https://aave.com/>
- [2] CertiKAlert 2025. CertiKAlert. <https://x.com/CertiKAlert>
- [3] Longfei Chen, Ruibin Yan, Taiyu Wong, Yiyang Chen, and Chao Zhang. 2025. SmartPoC: Generating Executable and Validated PoCs for Smart Contract Bug Reports. <https://arxiv.org/abs/2511.12993>
- [4] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, et al. 2020. SODA: A Generic Online Detection Framework for Smart Contracts. In *Proc. NDSS*.
- [5] Zhiyang Chen, Sidi Mohamed Beillahi, and Fan Long. 2024. FlashSyn: Flash Loan Attack Synthesis via Counter Example Driven Approximation. In *Proc. ICSE*. doi:[10.1145/3597503.3639190](https://doi.org/10.1145/3597503.3639190)
- [6] Gheorghe Comanici, Eric Bieber, Mike Schaeckermann, et al. 2025. Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities. <https://arxiv.org/abs/2507.06261>
- [7] Isaac David, Liyi Zhou, Kaihua Qin, Dawn Song, Lorenzo Cavallaro, and Arthur Gervais. 2023. Do you still need a manual smart contract audit? <https://arxiv.org/abs/2306.12338>
- [8] Isaac David, Liyi Zhou, Dawn Song, Arthur Gervais, and Kaihua Qin. 2025. Decompiling Smart Contracts with a Large Language Model. *arXiv:2506.19624* (2025).
- [9] dedaub 2025. dedaub. <https://app.dedaub.com/>
- [10] DeFiHackLabs 2025. SunWeb3Sec/DeFiHackLabs. <https://github.com/SunWeb3Sec/DeFiHackLabs>
- [11] defillama 2025. defillama. <https://defillama.com/>
- [12] Gelei Deng, Yi Liu, Yuekang Li, Kailong Wang, Ying Zhang, Zefeng Li, Haoyu Wang, Tianwei Zhang, and Yang Liu. 2024. MASTERKEY: Automated Jailbreaking of Large Language Model Chatbots. In *Proceedings 2024 Network and Distributed System Security Symposium*. doi:[10.14722/ndss.2024.24188](https://doi.org/10.14722/ndss.2024.24188)
- [13] erc20 2025. erc20. <https://eips.ethereum.org/EIPS/eip-20>
- [14] erc721 2025. erc721. <https://eips.ethereum.org/EIPS/eip-721>
- [15] ethereum/go-ethereum 2025. go-ethereum. <https://github.com/ethereum/go-ethereum>
- [16] etherscan 2025. etherscan. <https://etherscan.io/>
- [17] ethervm 2025. ethervm. <https://ethervm.io/decompile>
- [18] evm-tracing/custom-tracer 2025. Custom EVM tracer. <https://geth.ethereum.org/docs/developers/evm-tracing/custom-tracer>
- [19] foundry 2025. foundry-rs/foundry. <https://github.com/foundry-rs/foundry>
- [20] Arthur Gervais and Liyi Zhou. 2025. AI Agent Smart Contract Exploit Generation.
- [21] gpt5 2025. Introducing GPT-5 for developers. <https://openai.com/index/introducing-gpt-5-for-developers/>
- [22] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, Declarative Decomposition of Smart Contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. doi:[10.1109/ICSE.2019.00120](https://doi.org/10.1109/ICSE.2019.00120)
- [23] Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. 2022. Elipmoc: advanced decompilation of Ethereum smart contracts. *Proc. ACM Program. Lang.* 6, OOPSLA1 (April 2022). doi:[10.1145/3527321](https://doi.org/10.1145/3527321)
- [24] Fabio Gritti, Lorenzo Fontana, Eric Gustafson, Fabio Pagani, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. 2020. SYMBION: Interleaving Symbolic with Concrete Execution. In *2020 IEEE Conference on Communications and Network Security (CNS)*. doi:[10.1109/CNS48642.2020.9162164](https://doi.org/10.1109/CNS48642.2020.9162164)
- [25] Daya Guo, Dejian Yang, Haowei Zhang, et al. 2025. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature* 645, 8081 (Sept. 2025). doi:[10.1038/s41586-025-09422-z](https://doi.org/10.1038/s41586-025-09422-z)
- [26] Jinyao Guo, Chengpeng Wang, Xiangze Xu, Zian Su, and Xiangyu Zhang. 2025. Repoadit: An autonomous llm-agent for repository-level code auditing. *arXiv preprint arXiv:2501.18160* (2025).
- [27] Sujin Han, Jinseo Kim, Sung-Ju Lee, and Insu Yun. 2025. Automated Attack Synthesis for Constant Product Market Makers. *Proc. ACM Softw. Eng.* 2, ISSTA (June 2025). doi:[10.1145/3728872](https://doi.org/10.1145/3728872)
- [28] Heimdall 2025. Jon-Becker/heimdall-rs. <https://github.com/Jon-Becker/heimdall-rs>
- [29] Chao Hu, Wenhao Zeng, Yuling Shi, Beijun Shen, and Xiaodong Gu. 2026. In Line with Context: Repository-Level Code Generation via Context Inlining.
- [30] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. doi:[10.1109/ASE56229.2023.00181](https://doi.org/10.1109/ASE56229.2023.00181)
- [31] Nam Huynh and Beiyu Lin. 2025. Large Language Models for Code Generation: A Comprehensive Survey of Challenges, Techniques, Evaluation, and Applications.
- [32] Imen Jaoua, Oussama Ben Sghaier, and Houari Sahraoui. 2025. Combining Large Language Models with Static Analyzers for Code Review Generation. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE.
- [33] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2025. A Survey on Large Language Models for Code Generation. *ACM Trans. Softw. Eng. Methodol.* (July 2025). doi:[10.1145/3747588](https://doi.org/10.1145/3747588) Just Accepted.
- [34] Ziqiao Kong, Cen Zhang, Maoyi Xie, Ming Hu, Yue Xue, Ye Liu, Haijun Wang, and Yang Liu. 2025. Smart Contract Fuzzing Towards Profitable Vulnerabilities. *Proc. ACM Softw. Eng.* 2, FSE (June 2025). doi:[10.1145/3715720](https://doi.org/10.1145/3715720)
- [35] Sifis Lagouvardos, Yannis Bollanos, Neville Grech, and Yannis Smaragdakis. 2025. The Incredible Shrinking Context... in a Decomplier Near You. *Proc. ACM Softw. Eng.* 2, ISSTA (June 2025). doi:[10.1145/3728935](https://doi.org/10.1145/3728935)
- [36] Nguyet-Anh H. Lang, Eric Lang, Thanh Le-Cong, Bach Le, and Quyet-Thang Huynh. 2026. Perish or Flourish? A Holistic Evaluation of Large Language Models for Code Generation in Functional Programming.
- [37] Ye Liu, Yuqing Niu, Chengyan Ma, Ruidong Han, Wei Ma, Yi Li, Debin Gao, and David Lo. 2025. Towards Secure Program Partitioning for Smart Contracts with LLVM’s In-Context Learning. *arXiv preprint arXiv:2502.14215* (2025).
- [38] LoopTerminology 2025. LoopTerminology. <https://llvm.org/docs/LoopTerminology.html>
- [39] Openchain 2025. openchainxyz/openchain-monorepo. <https://github.com/openchainxyz/openchain-monorepo>
- [40] openrouter 2025. openrouter. <https://openrouter.ai/>
- [41] Panoramix 2025. eveem-org/panoramix. <https://github.com/eveem-org/panoramix>
- [42] Phalcon 2025. blocksec/phalcon. <https://blocksec.com/phalcon>
- [43] Kaihua Qin, Zhe Ye, Zhun Wang, Weilin Li, Liyi Zhou, Chao Zhang, Dawn Song, and Arthur Gervais. 2025. Enhancing Smart Contract Security Analysis with Execution Property Graphs. *Proc. ACM Softw. Eng.* 2, ISSTA (June 2025). doi:[10.1145/3728924](https://doi.org/10.1145/3728924)
- [44] Shouping Ren, Lipeng He, Tianyu Tu, Di Wu, Jian Liu, Kui Ren, and Chun Chen. 2025. LookAhead: Preventing DeFi Attacks via Unveiling Adversarial Contracts. *Proc. ACM Softw. Eng.* 2, FSE (June 2025). doi:[10.1145/3729353](https://doi.org/10.1145/3729353)
- [45] Hadis Rezaei, Mojtaba Eshghie, Karl Anderesson, and Francesco Palmieri. 2025. SoK: Root Cause of \$1 Billion Loss in Smart Contract Real-World Attacks via a Systematic Literature Review of Vulnerabilities. <https://arxiv.org/abs/2507.20175>
- [46] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2019. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In *Proc. NDSS*.
- [47] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes* 30, 5 (Sept. 2005). doi:[10.1145/1095430.1081750](https://doi.org/10.1145/1095430.1081750)
- [48] Zhang Sheng, Tan Kia Quang, Shen Wang, Shengchen Duan, Kai Li, and Yue Duan. 2025. Understanding and Characterizing Obfuscated Funds Transfers in Ethereum Smart Contracts. <https://arxiv.org/abs/2505.11320>
- [49] skylens 2026. skylens. <https://skylens.certik.com/>
- [50] SlowMist_Team 2025. SlowMist_Team. https://x.com/SlowMist_Team
- [51] Sunbeam So, Seongjoo Hong, and Hakjoon Oh. 2021. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In *Proc. USENIX Security*.
- [52] Joseph Spracklen, Raveen Wijewickrama, AHM Nazmus Sakib, Anindya Maiti, Bimal Viswanath, and Murtuza Jadiwala. 2025. We have a package for you! a comprehensive analysis of package hallucinations by code generating LLMs. In *Proceedings of the 34th USENIX Conference on Security Symposium*.
- [53] 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)*. <https://www.usenix.org/conference/usenixsecurity21/presentation/su>
- [54] Xing Su, Hanzhong Liang, Hao Wu, Ben Niu, Fengyuan Xu, and Sheng Zhong. 2025. DiSCO: Towards Decompiling EVM Bytecode to Source Code using Large Language Models. *Proc. ACM Softw. Eng.* 2, FSE (June 2025). doi:[10.1145/3729373](https://doi.org/10.1145/3729373)
- [55] 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*. doi:[10.1145/3597503.3639117](https://doi.org/10.1145/3597503.3639117)
- [56] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. LLM4Decompile: Decompiling Binary Code with Large Language Models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). doi:[10.18653/v1/2024.emnlp-main.203](https://doi.org/10.18653/v1/2024.emnlp-main.203)
- [57] TenArmorAlert 2025. TenArmorAlert. <https://x.com/TenArmorAlert>
- [58] tenderly 2025. tenderly. <https://dashboard.tenderly.co/>
- [59] Saad Ullah, Praneeth Balasubramanian, Wenbo Guo, Amanda Burnett, Hammond Pearce, Christopher Kruegel, Giovanni Vigna, and Gianluca Stringhini. 2025. From CVE Entries to Verifiable Exploits: An Automated Multi-Agent Framework for Reproducing CVEs. (2025). <https://arxiv.org/abs/2509.01835>
- [60] uniswap 2025. uniswap. <https://app.uniswap.org/>
- [61] usdt 2025. usdt. <https://tether.to/en/>
- [62] using_gpt5 2025. Learn best practices, features, and migration guidance for GPT-5. <https://platform.openai.com/docs/guides/latest-model>

- [63] Dabao Wang, Bang Wu, Xingliang Yuan, Lei Wu, Yajin Zhou, and Helei Cui. 2024. DeFiGuard: A Price Manipulation Detection Service in DeFi Using Graph Neural Networks. *IEEE Transactions on Services Computing* 17, 6 (2024). doi:10.1109/TSC.2024.3489439
- [64] 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*. doi:10.1145/3691620.3695526
- [65] WebKeyDaoIncident 2025. WebKeyDAO Hacked Incident. https://x.com/Phalcon_xyz/status/1900809936906711549
- [66] Hongbo Wen, Hanzhi Liu, Jiaxin Song, Yanju Chen, Wenbo Guo, and Yu Feng. 2024. FORAY: Towards Effective Attack Synthesis against Deep Logical Vulnerabilities in DeFi Protocols. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. doi:10.1145/3658644.3690293
- [67] Jiajing Wu, Kaixin Lin, Dan Lin, Bozhao Zhang, Zhiying Wu, and Jianzhong Su. 2025. Safeguarding Blockchain Ecosystem: Understanding and Detecting Attack Transactions on Cross-chain Bridges. In *Proceedings of the ACM on Web Conference 2025*. doi:10.1145/3696410.3714604
- [68] Siwei Wu, Zhou Yu, Dabao Wang, Yajin Zhou, Lei Wu, Haoyu Wang, and Xingliang Yuan. 2024. DeFiRanger: Detecting DeFi Price Manipulation Attacks. *IEEE Trans. Dependable Secur. Comput.* 21, 4 (July 2024). doi:10.1109/TDSC.2023.3346888
- [69] Yin Wu, Xiaofei Xie, Chenyang Peng, et al. 2024. AdvSCanner: Generating Adversarial Smart Contracts to Exploit Reentrancy Vulnerabilities Using LLM and Static Analysis. In *Proc. ASE*. doi:10.1145/3691620.3695482
- [70] 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 *Proceedings of the ACM on Web Conference 2025*. doi:10.1145/3696410.3714928
- [71] 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*. doi:10.1145/3650212.3652137
- [72] Jiacen Xu, Jack W. Stokes, Geoff McDonald, Xuesong Bai, David Marshall, Siyue Wang, Adith Swaminathan, and Zhou Li. 2024. AutoAttacker: A Large Language Model Guided System to Implement Automatic Cyber-attacks. (2024). <https://arxiv.org/abs/2403.01038>
- [73] Xiangzhe Xu, Zhuo Zhang, Zian Su, Ziyang Huang, Shiwei Feng, Yapeng Ye, Nan Jiang, Danning Xie, Siyuan Cheng, Lin Tan, and Xiangyu Zhang. 2024. Symbol Preference Aware Generative Models for Recovering Variable Names from Stripped Binary. (2024). <https://arxiv.org/abs/2306.02546>
- [74] Shuo Yang, Jiachi Chen, Mingyuan Huang, Zibin Zheng, and Yuan Huang. 2024. Uncover the Premeditated Attacks: Detecting Exploitable Reentrancy Vulnerabilities by Identifying Attacker Contracts. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. doi:10.1145/3597503.3639153
- [75] Sen Yang, Kaihua Qin, Aviv Yaish, and Fan Zhang. 2025. Insecurity Through Obscurity: Veiled Vulnerabilities in Closed-Source Contracts. (2025). <https://arxiv.org/abs/2504.13398>
- [76] Lei Yu, Zhirong Huang, Hang Yuan, Shiqi Cheng, Li Yang, Fengjun Zhang, Chenjie Shen, Jiajia Ma, Jingyuan Zhang, Junyi Lu, and Chun Zuo. 2025. Smart-LLaMA-DPO: Reinforced Large Language Model for Explainable Smart Contract Vulnerability Detection. *Proc. ACM Softw. Eng. 2, ISSTA* (June 2025). doi:10.1145/3728878
- [77] Bosi Zhang, Ningyu He, Xiaohui Hu, Kai Ma, and Haoyu Wang. 2025. *Following devils' footprint: towards real-time detection of price manipulation attacks*.
- [78] Jango Zhang. 2024. Combining GPT and Code-Based Similarity Checking for Effective Smart Contract Vulnerability Detection. *arXiv preprint arXiv:2412.18225* (2024).
- [79] Mengya Zhang, Xiaokuan Zhang, Yingqian Zhang, and Zhiqiang Lin. 2020. TXSPECTOR: Uncovering Attacks in Ethereum from Transactions. In *29th USENIX Security Symposium (USENIX Security 20)*. <https://www.usenix.org/conference/usenixsecurity20/presentation/zhang-mengya>
- [80] Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. 2025. LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation. *Proc. ACM Softw. Eng. 2, ISSTA* (June 2025). doi:10.1145/3728894
- [81] Juantao Zhong, Daoyuan Wu, Ye Liu, Maoyi Xie, Yang Liu, Yi Li, and Ning Liu. 2025. Detecting Various DeFi Price Manipulations with LLM Reasoning. (2025). <https://arxiv.org/abs/2502.11521>
- [82] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. 2023. Sok: Decentralized finance (defi) attacks. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE.
- [83] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. 2018. Frays: Reverse Engineering Ethereum's Opaque Smart Contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. <https://www.usenix.org/conference/usenixsecurity18/presentation/zhou>