# Thought Is All You Need: Smart Contract Vulnerability Detection with Thought-Augmented Large Language Model

CHAOYUAN PENG*, Zhejiang University, China
MUHUI JIANG, BlockSec, China
YAJIN ZHOU†, Zhejiang University, China and BlockSec, China
LEI WU, Zhejiang University, China and BlockSec, China

Smart contracts are self-executing agreements with code-defined terms enabling trustless blockchain transactions. Their immutability and control over significant financial assets make them attractive attack targets, with vulnerabilities potentially causing catastrophic financial losses. Large Language Models (LLMs) have revolutionized numerous domains with remarkable capabilities in code understanding and problem-solving. Despite these advancements, recent research reveals that LLMs still face significant limitations in accurately detecting complex vulnerabilities in smart contracts.

This disparity between the capabilities of LLMs and the stringent requirements of security analysis underscores the necessity for tailored methodologies to enhance LLM-based vulnerability detection strategies.

In this paper, we propose Synapse, the first smart contract vulnerability detection framework leveraging thought-augmented LLM and fine-grained analysis under focal context. Specifically, Synapse emulates security researchers' vulnerability discovery workflow, including vulnerability pattern learning, thought instantiation, reasoning, and verification. We employ a Buffer of Vulnerability Reasoning Thoughts (BoVRT) approach for LLMs to learn and apply vulnerability-specific reasoning to concrete contracts, improving detection accuracy. We also leverage specialized reasoning and code models to optimize different stages of the vulnerability detection process. To evaluate Synapse, we collected real-world on-chain contract incidents from security company alerts not covered by existing datasets. Synapse identified 117 previously undiscovered vulnerabilities in on-chain smart contracts, including one critical vulnerability that safeguarded assets totaling $30 million from potential losses.

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

Additional Key Words and Phrases: Smart contract security, vulnerability detection, large language models, multi-agent system

**ACM Reference Format:**
Chaoyuan Peng, Muhui Jiang, Yajin Zhou, and Lei Wu. 2026. Thought Is All You Need: Smart Contract Vulnerability Detection with Thought-Augmented Large Language Model. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE134 (July 2026), 24 pages. https://doi.org/10.1145/3808141

*Part of the work was accomplished when the author was a research intern at BlockSec.
†Corresponding author.

Authors' Contact Information: Chaoyuan Peng, Zhejiang University, Hangzhou, China, ret2happy@zju.edu.cn; Muhui Jiang, BlockSec, Hangzhou, China, jiangmuhui@blocksec.com; Yajin Zhou, Zhejiang University, Hangzhou, China and BlockSec, Hangzhou, China, yajin_zhou@zju.edu.cn; Lei Wu, Zhejiang University, Hangzhou, China and BlockSec, Hangzhou, China, lei_wu@zju.edu.cn.


ACM 2994-970X/2026/7-ARTFSE134
https://doi.org/10.1145/3808141

## 1 Introduction

Smart contracts form the bedrock of decentralized applications (dApps), leveraging the immutable and transparent nature of blockchain technology to enable trustless and efficient transactions, particularly within the finance industry. However, their susceptibility to vulnerabilities poses substantial financial risks. For instance, Defillama Hacks [20] reported over 100 incidents in 2024 alone, resulting in total losses of approximately $1.27 billion. Prominent examples, such as the Velocore hack [76] ($6.8 million) and the UwULend hack [63] ($19.3 million), underscore the critical need for smart contract vulnerability detection.

State-of-the-art static analysis frameworks [23] for smart contract vulnerabilities primarily identify flaws based on predefined control-flow or data-flow patterns, such as re-entrancy and integer overflows. However, these methods often lack a deep understanding of contract semantics, leading to high false positive rates. Similarly, existing contract fuzzing techniques [9, 27, 48, 55, 61] struggle to generate effective oracles due to the same limitation, hindering their ability to detect complex logic vulnerabilities. Furthermore, some approaches [5, 88] concentrate on specific vulnerability types, thereby offering limited protection against the diverse spectrum of critical vulnerabilities. Machine learning-based approaches [47, 57] are similarly limited by restricted vulnerability type coverage and insufficient semantic understanding of contracts, impeding their efficacy in detecting complex logic vulnerabilities. A comprehensive empirical analysis [89] of real-world vulnerabilities revealed that approximately 80% of exploitable vulnerabilities are classified as machine-unauditable bugs (MUBs), primarily due to this fundamental limitation in semantic comprehension. Furthermore, recent systematic evaluation [58] demonstrates a progressive decline in the effectiveness of existing vulnerability scanners, attributable to the rapid evolution of smart contract development practices and the emergence of novel attack vectors that exceed the detection capabilities of current tools.

Recent advancements in Large Language Models (LLMs), including GPT and DeepSeek, offer a promising avenue for automating smart contract vulnerability detection. Isaac et al. [16] demonstrated that while LLMs like GPT-4 and Claude-1.3 can identify attacks, they often produce high false positive rates (95%). GPTScan [68], which integrates GPT with static analysis, aims to detect logic vulnerabilities. However, its heavy reliance on static analyzers for vulnerability type filtering and lack of path sensitivity limit its detection accuracy to approximately 57% in large-scale projects. Although LLM enhancement techniques have been applied, existing studies [67, 87] demonstrate that approaches such as Retrieval-Augmented Generation (RAG) for integrating vulnerability knowledge attain only approximately 30% accuracy even when limited to identifying vulnerability types. Other LLM-driven works [65, 77] require historical transaction data, a prerequisite frequently unmet for newly deployed contracts or those under development. These methods fail to adequately replicate the complex research processes employed by security experts, leading to insufficient accuracy and overall capability.

**Key Challenges** While these approaches highlight the potential of LLMs in smart contract vulnerability detection, they fall short of accurately identifying vulnerabilities in real-world contracts due to several key challenges:

**First**, achieving high reasoning accuracy with long code contexts of multi-contracts remains a significant hurdle *(Challenge 1)*. Research [33, 39] indicates that mainstream LLMs typically utilize only 10-20% of the provided context, with performance degrading sharply as context length and task complexity increase. Although various reasoning enhancement techniques, such as Self-Consistency [79] (which improves reasoning through multi-round analysis and marginalization across different reasoning paths), have been proposed, methods like majority voting [40], a technique that selects the most frequent answer across multiple LLM inference passes, offer only marginal improvements.

**Second**, designing effective function tools for LLM to effectively retrieve code context is non-trivial *(Challenge 2)*. Human security researchers routinely employ documentation and sophisticated code analysis tools. However, current vulnerability detection frameworks [67] often fail to equip LLMs with the interactive tools (e.g., code insights, semantic search, documentation) necessary for analyzing large and complex contract projects. Furthermore, directly incorporating insights from static analyzers, such as redundant call graphs and standard library implementations, can potentially confuse LLMs and degrade performance [40]. While leading code editors [11, 14] (e.g., Cursor, Windsurf) provide LLMs with insightful tools for accurate code generation, they lack the specialized security-domain tools, such as vulnerability-pattern-aware code search, inter-contract state tracking, and exploit scenario simulation, required for effective vulnerability detection.

**Third**, pre-trained LLMs possess limited knowledge of smart contract vulnerabilities *(Challenge 3)*. LLMs trained on data up to specific cutoff dates are inherently unaware of emerging vulnerability patterns. Moreover, general-purpose models lacking specialized security training often struggle to learn comprehensive contract vulnerability types. Consequently, providing LLMs with complete and structured vulnerability reasoning patterns is crucial.

**Our Approach** To address these challenges, we introduce Synapse, a thought-augmented multi-agent framework for smart contract vulnerability detection that emulates the research processes of security experts. The core insight underpinning Synapse is that security researchers exhibit common characteristics: *analogical reasoning regarding vulnerability types, focused contextual research, and iterative "reasoning-acting" loops*.

To address *Challenge 1*, we employ a focal context mechanism (detailed in Section 4.2) during LLM analysis to enhance reasoning accuracy within limited context windows. Unlike existing works [80] that typically provide entire contract code to LLMs, leading to performance degradation from excessive context length, Synapse supplies fine-grained code context at various levels of focus when exploring a vulnerability hypothesis. For Solidity contracts, we preprocess code at the Abstract Syntax Tree (AST) level, identifying key functions while excluding standard library implementations. This AST-level preprocessing mitigates compatibility issues arising from compilation failures and facilitates extension to contracts written in other programming languages.

To tackle *Challenge 2*, we implement multiple specialized agents equipped with powerful security-domain tools: Developer, Researcher, Auditor, and Verifier. The Developer agent extracts code insights for the entire contract project. The Researcher agent retrieves vulnerability patterns from a curated thought collection and instantiates concrete reasoning paths within the focal context. The Auditor agent performs vulnerability reasoning and detection based on these concrete vulnerability thoughts, utilizing security-domain tools. The Researcher and Auditor agents coordinate to transform high-level patterns into concrete reasoning steps. Finally, the Verifier agent validates identified vulnerabilities. We have designed customized security tools, including semantic search and code differential analysis, which enable accurate search functionalities and self-directed reasoning-acting loops without requiring manual intervention. Our evaluation demonstrates that these semantic tools improve recall by 44.5% on real-world datasets.

To address *Challenge 3*, we leverage a "Buffer of Vulnerability Reasoning Thoughts" (BoVRT), a collection of structured, reusable reasoning paths distilled from real-world vulnerability analyses, encompassing various contract types. In contrast to existing approaches that retrieve raw vulnerability reports, Synapse retrieves high-level vulnerability thought templates and instantiates them for specific contract code. Unlike standard RAG, which directly injects retrieved documents into the LLM prompt, BoVRT performs a multi-step instantiation process that transforms abstract reasoning templates into concrete, code-specific analytical workflows (detailed in Section 4.3.1). We utilize a cost-effective model to distill and generate vulnerability patterns from over 14,000 real-world audit

reports, constructing a vector database containing embedded hierarchical vulnerability thoughts. For each target contract, Synapse retrieves relevant patterns from this database and instantiates concrete reasoning paths tailored to the specific code. These thoughts guide the agents in reasoning about and verifying potential vulnerabilities.

We implemented Synapse and evaluated its performance on real-world smart contracts. Our results demonstrate that Synapse detects 71.9% of vulnerabilities in these datasets, achieving a 3× improvement compared to state-of-the-art LLM-driven tools like GPTScan [68]. Furthermore, our buffer of thoughts approach enhances recall by 36.3%, outperforming traditional analysis tools by at least 2×. Synapse successfully identified 117 previously undiscovered vulnerabilities in on-chain smart contracts, including one critical vulnerability that safeguarded assets totaling $30 million from potential losses.

In summary, our contributions are as follows:

- **Novel Analysis Approach under Focal Context.** We propose a flexible focal context mechanism that provides LLMs with concise and structured code semantics, enabling effective utilization of limited token context and thereby improving reasoning accuracy.
- **Multi-Agent Enhancement with Semantic-aware Tools.** We implement semantic tools that empower agents to mimic the workflows of security researchers during vulnerability discovery in large and complex smart contract projects.
- **Buffer of Vulnerability Reasoning Thoughts.** We introduce a buffer of vulnerability reasoning patterns that enables LLMs to explore diverse vulnerabilities based on contract semantics. By distilling high-level vulnerability patterns from over 14,000 real-world audit reports, we have created a knowledge base that specialized agents can instantiate into concrete reasoning paths, effectively guiding the vulnerability detection process.
- **Effective Prototype System and New Findings.** We present Synapse, an effective multi-agent system for smart contract vulnerability detection evaluated on real-world contracts. Synapse identified 117 previously undiscovered vulnerabilities in on-chain smart contracts, including a critical vulnerability that protected assets valued at **$30 million** from potential losses.

## 2 Background

This section briefly explains the key terms (smart contract vulnerability and large language models) used throughout this paper.

### 2.1 Smart Contract Vulnerabilities

Smart contracts are self-executing programs deployed on blockchains, enabling the autonomous implementation of programmable business logic. Mainstream smart contracts are written in Solidity, a high-level programming language designed for the Ethereum blockchain. Major smart contract applications include decentralized exchanges (DEXs) [83], decentralized autonomous organizations (DAOs) [38], lending platforms [84], yield farming [82], and stablecoins [3]. The immutable nature of contracts and their control over significant financial assets make them attractive targets for attackers, where vulnerabilities can lead to catastrophic financial losses. An empirical study [89] systematically investigated real-world vulnerabilities from audit findings, categorizing exploitable bugs into two types: machine-auditable bugs (MABs) and machine-unauditable bugs (MUBs). Due to the lack of semantic understanding of domain-specific properties, more than 80% of exploitable bugs are machine-unauditable. MABs include assertion failures, arbitrary writes, block-state dependencies, and integer bugs, while MUBs include price oracle manipulation, erroneous accounting, ID uniqueness violations, and privilege escalation.

```
1  function transferToUnoccupiedPlot(uint256 tokenId, uint256 plotId) external {
2      ToilerState memory _toiler = toilerState[tokenId];
3      uint256 oldPlotId = _toiler.plotId;
4      uint256 totalPlotsAvail = _getNumPlots(_toiler.landlord);
5      ... // checks for toiler and plotID
6      toilerState[tokenId].latestTaxRate= plots[_toiler.landlord].currentTaxRate;
7      // missing: toilerState[tokenId].plotId = plotId
8      plotOccupied[_toiler.landlord][oldPlotId] = Plot({
9          occupied: false,
10         tokenId: 0
11     });
12     emit FarmPlotLeave(_toiler.landlord, tokenId, oldPlotId);
13     emit FarmPlotTaken(toilerState[tokenId], tokenId);
14 }
```

Fig. 1. Motivating Example. The logic vulnerability (line 7) causes inconsistency in plot tracking due to the missing update of the plot ID during the transfer operation.

### 2.2 Large Language Models for Vulnerability Detection

Large language models (LLMs) [45], such as GPT [50] and DeepSeek [28], are advanced machine learning models trained via self-supervised learning on vast textual datasets. These models exhibit remarkable proficiency in natural language processing and related domains, leading to substantial research in this area.

LLMs have shown proficiency in analyzing program semantics [44]. By utilizing their code comprehension capabilities, LLMs can identify vulnerabilities with zero-shot prompting [90], where examples of vulnerabilities are not needed for detection. Detection approaches are further enhanced by advanced techniques such as prompt engineering [41] and retrieval-augmented generation (RAG) [26]. Du et al. [22] applied chain-of-thought (CoT) prompting for smart contract security audits and found that GPT-4 performed poorly in detecting vulnerabilities, with a low recall of 37.8%. Shimmi et al. [59] demonstrated that few-shot prompts with additional domain-specific knowledge optimize LLM performance for vulnerability detection. Furthermore, Daneshvar et al. [15] found that vulnerability detection can be improved by incorporating RAG-based augmentation. Different models exhibit varying performance [69] across different tasks (e.g., code generation, mathematical reasoning, language understanding). Therefore, we integrate different models to build Synapse.

## 3 Motivation

In this section, we build on the discussion from Section 1 and present a real-world vulnerability to highlight the necessity of LLMs for automated smart contract vulnerability detection.

Figure 1 showcases a critical vulnerability in the Munchables project [70], stemming from a missing contract state update. The LandManager contract implements a staking mechanism, enabling users to stake tokens on others' plots for rewards. The `transferToUnoccupiedPlot` function supports token transfers between plots. However, a severe flaw exists: although the token is moved to a new plot, the `plotId` field in the `ToilerState` struct remains unchanged. This omission causes a state inconsistency, as the internal records incorrectly reflect the token's original location post-relocation.

This state inconsistency can be exploited because when the total number of plots is reduced, stale records allow another user to stake in an already-occupied plot.

Static analysis tools, while capable of detecting certain fixed patterns for missing state updates, often fail to identify which functions should update specific states. Likewise, advanced fuzzers [61] miss this vulnerability due to their reliance on non-semantic oracles, which only trigger alerts under predefined violation conditions.

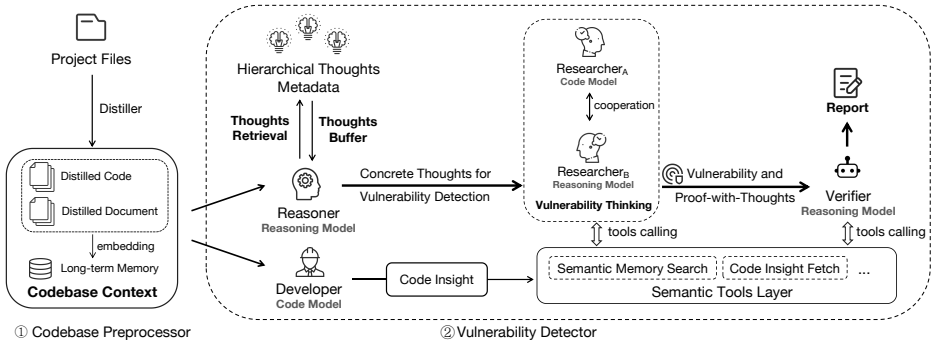


Fig. 2. Overall workflow of Synapse.

In contrast, Synapse detects this vulnerability by retrieving a *targeted thought* template addressing state inconsistency in staking mechanisms from the BoVRT (Section 4.3.1). The **Reasoner** instantiates this template with concrete references (e.g., `transferToUnoccupiedPlot`, `plotOccupied`), and the **Researcher** agents employ focal context to trace the missing `plotId` update across the function body and its callees. The **Verifier** agent confirms the finding by validating that the evidence accurately reflects the code's behavior.

## 4 Design and Implementation

This section introduces the design of Synapse. Figure 2 depicts the overall workflow, which includes the codebase preprocessor (①) and the vulnerability detector (②). The vulnerability detector consists of the **Reasoner**, **Developer**, **Researcher**, and **Verifier** agents. First, we distill the contract codebase into code and documentation files, embed them into a vector database, and generate the codebase context (Section 4.1). Subsequently, the **Reasoner** agent generates *instantiated thoughts* for vulnerability detection via a buffer of thoughts. Meanwhile, the **Developer** agent generates code insights containing project invariants for the semantic tools. Afterwards, two **Researcher** agents leverage these instantiated vulnerability thoughts and semantic tools to identify vulnerabilities under different focal contexts. Notably, Synapse uses different models (a code model and a reasoning model) to back the two **Researcher** agents for effective cooperation. Finally, the **Verifier** agent validates the findings according to the vulnerability thoughts and generates the final report.

### 4.1 Codebase Preprocessor

Smart contract projects comprise multiple files, including contract code, standard libraries, dependencies, and documentation. However, not all files are relevant to vulnerability detection. Working with irrelevant files or code increases LLM context length, degrades accuracy [33, 39], and can introduce false positives. Therefore, we propose a *Distiller* (preprocessor) to preprocess the codebase and generate the focused code context.

*4.1.1 AST Parsing.* Before semantic search or code exclusion, an overview of the codebase is necessary. We perform AST-level parsing to extract contract details. We operate at the Abstract Syntax Tree (AST) [46] level, rather than other semantically aware levels (e.g., intermediate representation [64]), because alternative approaches require complete project compilation. This requirement would prevent extension to many use cases, including the decompiled code of closed-source contracts and contracts in other languages.

State-of-the-art LLM-based tools, such as GPTScan [68], heavily rely on Solidity compiler results for static analysis. This approach fails to analyze 17 of the Top 200 smart contract projects and 28

of 100 Code4Rena audited projects due to compilation failures or missing library dependencies. In contrast, SYNAPSE avoids these limitations by operating at the AST level, enabling the analysis of open-source projects and closed-source contracts via decompiled code.

We leverage ANTLR [53] to generate and refine the AST according to Solidity features. Since ANTLR performs purely syntactic parsing without requiring compilation, it enables analysis of contracts with missing dependencies. The refinement resolves `Import` and `Using` statements, facilitating type inference. By traversing the resulting AST, we extract function declarations and code bodies for subsequent vulnerability detection.

*4.1.2 Standard Implementation Exclusion.* Since most standard libraries and dependencies are irrelevant to vulnerability detection, we exclude them. We first collect implementations of well-known standard libraries and protocols across different versions. Standard libraries are sourced from official repositories like OpenZeppelin and Solmate. We also collect implementations of established protocols such as Uniswap V2/V3 and Compound. As these implementations are frequently used, manually audited, and battle-tested, we consider them reliable and exclude them from the detection process.

Existing work [68] uses directory names and function signatures as exclusion rules, which is neither accurate nor comprehensive. For example, an ERC20 token might reuse OpenZeppelin library code in functions (e.g., `balanceOf`, `approve`). These functions might not be excluded if their signatures are not whitelisted. Therefore, we propose a fine-grained, function-level exclusion mechanism that considers function bodies. After extracting functions from the collected standard implementations, we generate standard AST metadata for all functions. We exclude functions whose signatures and AST structures match standard implementations.

*4.1.3 Codebase Distillation.* After excluding standard implementations, we further exclude dependencies based on mainstream contract development toolkit structures (e.g., Foundry [25], Hardhat [49]). Since contract projects may contain documentation that can aid LLM understanding, we extract document files (e.g., plain text, Markdown) and split them into chunks based on document structure. For instance, we split Markdown top-level sections into chunks to improve document search accuracy.

*4.1.4 Long-Term Memory Generation.* Similar to security experts who rely on semantic memory to recall critical code snippets, we leverage LLM long-term memory technologies [35, 78] to generate codebase memory that facilitates semantic code search during vulnerability detection.

Specifically, we use CodeBERT [24] to generate embeddings [42] of functions (from Section 4.1.1) and document chunks (from Section 4.1.3), and store them in a vector database (Chroma [10]). This enables semantic search via k-nearest neighbors to retrieve the most relevant code snippets as long-term memory. This memory is generated offline as a one-time effort and reused during detection. We use RAG [35] to access this long-term memory in the tools layer. It is important to distinguish this codebase-level RAG, which retrieves code snippets and documentation to augment agent queries, from the Buffer of Vulnerability Reasoning Thoughts (BoVRT) described in Section 4.3.1, which retrieves and instantiates structured vulnerability reasoning templates rather than raw documents.

### 4.2 Semantic Tools Layer

Since human security researchers excel at utilizing domain-specific toolchains to validate their thoughts, we provide semantic tools for agents to validate their thoughts as well. To enhance the attention and thinking quality of agents, we propose a focal context to guide them to focus on the most relevant code segments.

**Prompt Template for Code Insight Generation**

**System:** You are the best programmer and the developer in the world. You'll be provided with a set of contract code, along with the basic code structure. Your task is to generate the business flow for this contract and strictly following the below instructions:
1. Initiate analysis from leaf contracts, performing function-level control flow simulation for multiple times and select the most reliable one.
2. When encountering unfamiliar function calls, retrieve the code through the provided tools to reconstruct complete execution paths.
3. When encountering external calls you fail to resolve, you should skip them in the final graph.
4. ALWAYS take care of the incoming and the outgoing edges of each function.
5. ALWAYS follow the tool call schema exactly as provided.
...

**Basic Code Structure**
There are {file_numbers} files in the codebase: {file_list}.
Excluding standard implementations, we provide the structured function mapping:
{file_path: {contracts_list: functions_list }}.

**Provided Tools**
<get_function_code/> <get_contract_code/>
<get_contract_inheritance/> <grep_code_search/>
<fuzzy_search_document/>

Fig. 3. Prompt template for code insight generation. Tools schema and output formats are omitted for simplicity.

```
contract UniswapV2Pair is IUniswapV2Pair, UniswapV2ERC20 {
    address public factory;                                          f_f + f_m + f_i + f_v
    address public token0;
    address public token1;

    uint112 private reserve0;
    uint112 private reserve1;
    uint private unlocked = 1;

    function _safeTransfer(address token, address to, uint value) private {   f_f + f_m + f_i
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(SELECTOR, to, value));
        require(success && (data.length == 0 || abi.decode(data, (bool))), 'TRANSFER_FAILED');
    }
    modifier lock() {                                                f_f + f_m
        require(unlocked == 1, 'UniswapV2: LOCKED');
        unlocked = 0;
        _;
        unlocked = 1;
    }
    function skim(address to) external lock {                        f_f
        address _token0 = token0;
        address _token1 = token1;
        _safeTransfer(_token0, to, IERC20(_token0).balanceOf(address(this)).sub(reserve0));
        _safeTransfer(_token1, to, IERC20(_token1).balanceOf(address(this)).sub(reserve1));
    }

}
```

Fig. 4. Focal context example for the `skim` function. The code is simplified from the UniswapV2Pair Contract.

*4.2.1 Code Insight Generation.* We utilize the **Developer** agent to generate *Code Insight*, i.e., the business flow with a simplified inter-function call tree. We observe that LLMs perform well as "compilers" [13] and can generate concise call trees. Therefore, instead of using static analyzers to extract call graph metadata, we leverage the code model, armed with code tools, to extract the call graph metadata. This design choice is motivated by two key observations. First, static analyzers produce call graphs that include extensive standard library calls and framework-internal invocations, which introduce significant noise that can confuse LLMs during vulnerability analysis [40]. Second, LLM-generated call graphs inherently capture semantic relationships between functions, such as business flow dependencies, that purely syntactic call graph extraction cannot represent. In practice, we utilize the Qwen2.5-Coder model [30] to generate the inter-function call graph for each contract using the prompts shown in Figure 3.

*4.2.2 Focal Context.* When feeding core information of relevant code to agents, an effective way to represent the code is needed. Through function-level code analysis that prioritizes critical functions, LLMs can effectively [39] leverage their short-term memory (i.e., the prompt). Additional contextual information for the focal function can provide useful clues and improve reasoning accuracy. We categorize the focal context into the following four types:

- $f_f$: The function body and declaration code of the focal function.
- $f_m$: The modifier code of the focal function.
- $f_i$: All internal functions callable by the focal function. We return results with a completeness check (verifying that all transitively called internal functions are included) according to the function call graph produced in Section 4.2.1.
- $f_v$: All private and public state variables in the contract of the focal function.

During the vulnerability analysis process, agents dynamically select the appropriate focal context granularity based on the current reasoning requirements. The **Researcher** agents begin with the function body ($f_f$) and progressively expand the context as needed. When the reasoning steps in the *instantiated thoughts* reference modifier-related logic (e.g., access control checks), agents request the modifier context ($f_m$). When internal function calls require examination, agents request $f_i$. When state variable tracking is necessary for understanding data flow across functions, agents request $f_v$. Figure 4 demonstrates the composability across different focal context types for the `skim` function in the UniswapV2Pair contract. We generate each type of focal context with the refined AST structure extracted in Section 4.1.1.

*4.2.3 Semantic Memory Search.* We leverage the long-term memory established in Section 4.1.4 to enable semantic search capabilities for agents. Unlike traditional keyword-based approaches, semantic search understands the conceptual meaning behind queries. For instance, when investigating potential price manipulation vulnerabilities, **Researcher** agents can query for "price oracle functions" and retrieve semantically relevant results (such as `getBNBPrice`, etc.) that might not contain explicit keywords in the query. This reveals vulnerable patterns conventional methods miss.

Specifically, when an agent requests a semantic search query, we use the same embedding model mentioned in Section 4.1.4 to generate the query embedding. We retrieve top-$k$ matches, extracting function focal context from code snippets. We provide options for agents to choose their preferred granularity (i.e., $f_f$, $f_m$, $f_i$, $f_v$, or combinations of these) of the focal context for the code result.

### 4.3 Vulnerability Detection

Our approach employs an LLM-driven multi-agent framework for codebase vulnerability detection. The framework comprises three key components: focal context, guiding detection by identifying relevant code segments; a semantic tools layer (Section 4.2), enhancing accuracy through specialized code analysis; and a buffer of thoughts (Section 4.3.1), improving detection by maintaining reasoning chains and intermediate findings across multiple analysis steps.

We categorize Machine Unauditable Bugs (MUBs) into two types: checklist-based and invariant-based vulnerabilities. The former relies on checklists derived from auditors, security researchers, and real-world vulnerability reports, while the latter focuses on whether code invariants can be violated. Correspondingly, we classify vulnerability reasoning thoughts into two categories:

- *Targeted thoughts* encode multi-step reasoning procedures for detecting specific vulnerability patterns. Each is distilled from a real-world audit report and contains a direct question, a sequence of reasoning steps, and the required function signatures and state variables.

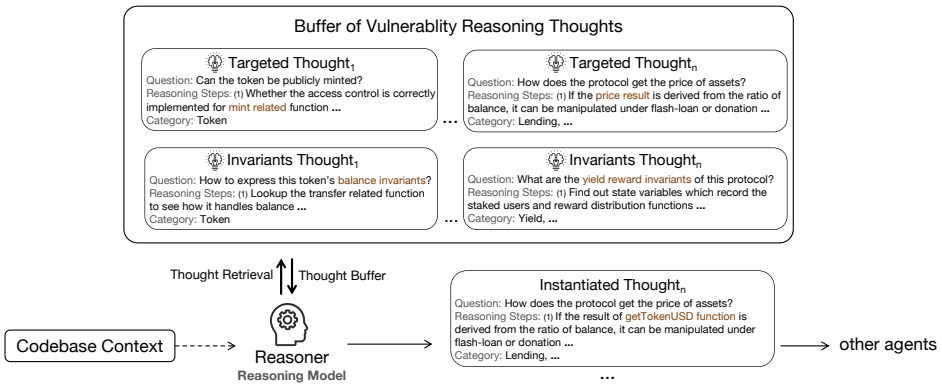


Fig. 5. Pipeline of the buffer of vulnerability reasoning thoughts. The Reasoner agent retrieves targeted and invariant thoughts from the buffer. Using codebase context, the Reasoner agent generates concrete *instantiated thoughts* for other agents.

Table 1. Vulnerability classification taxonomy employed by Synapse. Coverage indicates whether the type is addressed by *targeted thoughts* (T), *invariant thoughts* (I), or both (T+I).

| Category | Sub-types | MAB/MUB | Coverage |
|---|---|---|---|
| Access Control (AC) | Privilege escalation, missing authorization | MUB | T |
| Price Manipulation (PM) | Oracle manipulation, flash loan attacks | MUB | T+I |
| Calculation Error (CE) | Rounding errors, precision loss | MUB | T |
| Reentrancy (RE) | Cross-function, cross-contract | MAB/MUB | T |
| Logic Flaw (LF) | State inconsistency, business logic errors | MUB | T+I |

- *Invariant thoughts* guide agents to derive and verify contract-level invariants (e.g., "the sum of all user balances must equal the total supply") rather than following a fixed checklist. This category is effective for detecting novel logic flaws absent from the targeted thought corpus.

**Vulnerability Classification Taxonomy.** Table 1 presents the taxonomy derived from the 14,000+ audit reports via LLM-assisted classification with manual verification. Each category maps to *targeted thoughts* (T), *invariant thoughts* (I), or both (T+I). Novel patterns not represented in the corpus may require incremental expansion of the thought buffer.

The workflow is as follows: First, the *Reasoner* agent retrieves vulnerability thought templates and instantiates them based on the codebase context. Subsequently, two *Researcher* agents collaborate to determine if vulnerabilities exist under different focal contexts. Finally, the *Verifier* agent verifies these thoughts and generates the final report.

*4.3.1 Buffer of Vulnerability Reasoning Thoughts (BoVRT).* State-of-the-art reasoning models often lack structured, domain-specific vulnerability knowledge, causing them to overlook many vulnerability types during zero-shot prompting. The original Buffer of Thoughts [86] utilizes high-level thoughts from problem-solving processes across various tasks to enhance LLM accuracy, efficiency, and robustness. Inspired by this, we propose the Buffer of Vulnerability Reasoning Thoughts (BoVRT) to provide agents with vulnerability-specific knowledge and enhance their in-context learning capabilities.

**Generation of BoVRT** Vulnerability reasoning is crucial for security researchers to identify smart contract weaknesses. These patterns are documented in professional resources such as audit reports, vulnerability write-ups, and post-mortem analyses. We collected over 14,000 real-world vulnerability reports from platforms like Code4rena, Sherlock, and Immunefi. We used few-shot prompting (Figure 6) to distill *targeted thoughts* from these reports using the DeepSeek-V3 model.

**Prompt for Vulnerability Reasoning Thoughts Generation**

**System:** You are the best security researcher in the world. You'll be provided with a vulnerability report from other security researchers. Your task is to generate the vulnerability reasoning thoughts for this report and strictly following the below instructions:
1. First, you need to identify the target contract category from the report. The possible categories are Lending, Dexes, Yield, Cross-chain, Token, Governance, Others.
2. Second, by following the thinking of report, you need to extract the detailed reasoning thoughts to find out the vulnerability without missing any details. You SHOULD focus on the business flow and the subtle code pattern.
3. You should simulate the reasoning process using the extracted thoughts, and if you cannot discover the vulnerability by exactly following your extracted thoughts, you should re-extract the reasoning thoughts until you can successfully reproduce the vulnerability finding.
4. After verify the extracted thoughts, you should simplify and distill the thoughts by removing the concrete code related with this contract.

**Vulnerability Report**
{vulnerability_report_content_in_plain_text}

**Provided Tools**
<get_url_content/>

Fig. 6. Prompt template for vulnerability reasoning thoughts generation from the given vulnerability report. Output formats are omitted for simplicity.

Table 2. Comparison of BoVRT and RAG approaches.

| Dimension | RAG | BoVRT |
|---|---|---|
| Retrieved Content | Raw documents or vulnerability reports | Distilled reasoning thought templates with multi-step logic |
| Post-retrieval | Direct concatenation into prompt | Three-step instantiation into executable reasoning paths |
| Usage Pattern | Supplementary context for generation | Guides multi-agent reasoning as analytical workflows |

Specifically, as instruction #3 in the distillation prompt (Figure 6) mandates, the LLM simulates the reasoning process using the extracted thoughts. If the LLM cannot reproduce the vulnerability finding by following the extracted reasoning steps, it re-extracts the thoughts iteratively until successful reproduction is achieved. This self-verification loop ensures that each thought template preserves sufficient detail to guide vulnerability discovery.

The distillation process yielded **1,247** thought templates, of which **81%** passed self-verification without re-extraction. A manual spot-check of 100 randomly sampled templates achieved a **92%** accuracy rate. We categorized contracts into seven types (Lending, DEXs, Yield, Cross-chain, Token, Governance, and Others) to narrow the retrieval scope.

**BoVRT versus RAG** We define a *thought template* as a structured tuple ($contract\_type$, $direct\_question$, $reasoning\_steps$, $state\_variables$, $function\_signatures$). Unlike RAG, which retrieves raw documents and concatenates them, BoVRT retrieves *distilled reasoning templates* and performs a three-step instantiation to produce executable reasoning paths tailored to the target codebase (Table 2). Disabling BoVRT (SYNAPSE$_b$) reduces recall from 0.719 to 0.462 (Table 5).

**Pipeline of BoVRT.** Figure 5 illustrates the BoVRT pipeline, and Algorithm 1 formalizes the procedure. We employ CodeBERT [24] embeddings stored in a Chroma vector database. The *Reasoner* agent retrieves the top-$k$ ($k$=10) thought templates via cosine similarity, filtered by the target contract type. The instantiation process encompasses three steps:

(1) **Concrete Function Filling.** The *Reasoner* agent employs semantic search to map abstract function references to actual implementations in the target contract.
(2) **Concrete State Variable Extraction.** Using $f_v$ of the focal context, the agent grounds abstract variable references to specific storage slots.
(3) **Business Flow Adaptation.** The agent leverages *Code Insight* to incorporate simplified key business flows into the reasoning steps.

**Algorithm 1:** Thoughts Retrieval and Instantiation

**Input:** Target contract $C$, code context $CC$, thought buffer $\mathcal{B}$
**Output:** Instantiated thoughts $\mathcal{T}_{inst}$

$type \leftarrow$ ClassifyContractType($CC$);
$query \leftarrow$ GenerateQuery($type$, Summary($CC$));
$\mathcal{T}_{raw} \leftarrow$ RetrieveTopK($\mathcal{B}, query, k$=10);
$\mathcal{T}_{inst} \leftarrow \emptyset$;
**foreach** $t \in \mathcal{T}_{raw}$ **do**
    $t_{func} \leftarrow$ ConcreteFunctionFilling($t, C$);
    $t_{var} \leftarrow$ StateVariableExtraction($t_{func}, f_v$);
    $t_{flow} \leftarrow$ BusinessFlowAdaptation($t_{var}, CC$);
    $\mathcal{T}_{inst} \leftarrow \mathcal{T}_{inst} \cup \{t_{flow}\}$;
**end**
**return** $\mathcal{T}_{inst}$;

*4.3.2 Vulnerability Thinking.* The *Developer* agent generates *Code Insight*, and the *Reasoner* agent produces *instantiated thoughts* through the three-step process described in Section 4.3.1. Two *Researcher* agents then identify vulnerabilities based on these instantiated thoughts.

We noticed that depending on a single model restricts its effectiveness and prevents it from capitalizing on the strengths of other models to address its weaknesses. Therefore, as shown in Figure 2, we use two specialized models to leverage their complementary strengths. The code model excels at code analysis, while the reasoning model performs better in logical tasks, such as mathematical evaluation. Through collaboration, each *Researcher* agent benefits from the other's findings and critiques the other's work. Initially, both $Researcher_A$ (code model) and $Researcher_B$ (reasoning model) receive *instantiated thoughts* and use semantic tools to find potential vulnerabilities. Subsequently, each agent reviews the other's findings, offers criticism, and conducts another round of analysis based on this criticism.

During the thinking process, *Researcher* agents use semantic tools (Section 4.2) to mimic human researchers. Each instantiated thought contains a *direct question* as the entry point. If the answer is unsatisfactory, the agent omits this vulnerability type. Otherwise, agents follow the reasoning steps and generate *proofs-of-thoughts*, i.e., step-by-step verification evidence grounded in the actual code.

*4.3.3 Vulnerability Verification.* The six verification criteria were derived from hallucination patterns observed during preliminary experiments. The *Verifier* agent must answer Yes/No to each item; a vulnerability is included in the final report only if all answers are Yes.

(1) Is each proof-of-thoughts factually accurate and non-contradictory? [*factuality*]
(2) Does each proof-of-thoughts logically connect to the final decision? [*factuality*]
(3) Are vulnerabilities based on malicious miner/block builder behavior excluded? [*false positive*]
(4) Does each proof-of-thoughts directly relate to the identified vulnerability? [*faithfulness*]
(5) Does each proof-of-thoughts avoid assumptions beyond the provided information? [*faithfulness*]
(6) Is all analysis strictly limited to the given code, without external assumptions? [*faithfulness*]

## 4.4 Implementation

We implemented SYNAPSE in approximately 4,600 lines of Python code. Key implementation details are highlighted below.

*4.4.1 Codebase Preprocessor.* As described in Section 4.1, we use ANTLR [53] to parse smart contract code and generate the initial abstract syntax tree (AST). We refine the AST by resolving types and dependencies to produce $AST_{refined}$ and extract function information. For long-term memory generation, we utilize the CodeBERT [24] model to embed function information into a vector database using Chroma [10], along with the metadata. Rather than using online embedding models like *text-embedding-3*, we employ a local embedding model to eliminate API call costs and

Table 3. Datasets for evaluation. Vulnerabilities are categorized by *primary* type; some may involve multiple issues. *Num* indicates the number of vulnerabilities per dataset. Abbreviations: AC (AccessControl), PM (Price Manipulation), CE (Calculation Error), RE (Reentrancy), LF (Logic Flaw).

| Dataset | Vulnerability Types | | | | | Num |
|---|---|---|---|---|---|---|
| | AC | PM | CE | RE | LF | |
| Incidents | 61 | 163 | 8 | 3 | 14 | 249 |
| DeFiHacks | 3 | 8 | 1 | - | 5 | 17 |
| Top-100 | - | - | - | - | - | 100 |

enhance embedding efficiency. This approach enables the preprocessor to efficiently handle most smart contract projects while maintaining high performance during vulnerability detection.

*4.4.2 Vulnerability Detector.* We implement the multi-agent architecture and tool-calling adaptation using the LangChain framework [34]. For the reasoning model, we employ DeepSeek-R1-Distill-Qwen-32B [18], which augments the foundational Qwen2.5-32B model [54] through distillation techniques derived from the DeepSeek R1 model series. For the code model, we utilize DeepSeek-V3 [19], which has demonstrated superior performance [37] in code-related tasks.

We set the `Temperature` parameter to 0 for the code model to minimize randomness in the analysis. For all other parameters across all models, we maintain their default values.

## 5 Evaluation

*Datasets.* To comprehensively measure detection capabilities, we collected three datasets from real-world contracts, as summarized in Table 3.

First, we collected real-world on-chain attack incidents identified through continuous monitoring by security teams including BlockSecTeam [4], TenArmor [71], and Decurity [17]. These incidents are representative, involving high-severity vulnerabilities that caused significant financial losses. We excluded incidents related to rug pulls [66] or closed-source projects, resulting in 249 incidents. We extracted core vulnerable functions from these incidents as input for the *Incidents* dataset. Second, we collected the DeFiHacks dataset, comprising 17 real-world vulnerabilities with documented security incidents causing financial losses exceeding $100,000 per incident. Ground-truth vulnerabilities in these datasets were verified through post-mortem reports and alerts by established security firms, with community validation. Notably, each project in these datasets comprises multiple contract files (an average of 7 units in *Incidents* and 5 in *DeFiHacks*). The majority of vulnerabilities are multi-base unit (MBU) vulnerabilities [56] that span multiple contracts or code units: 187 out of 249 (75.1%) in *Incidents* and 83% in *DeFiHacks*. These cross-contract vulnerabilities require holistic reasoning across multiple code units, a capability that Synapse's focal context and semantic search are specifically designed to provide. Finally, we selected the top 100 smart contracts with the highest transaction volumes on Ethereum as the *Top-100* dataset. These contracts have been widely used and battle-tested in production environments over extended periods, making them representative specimens of secure and reliable smart contracts.

*Experiments Setup.* We used the official OpenAI API [52] to interface with proprietary GPT-series models and the SiliconFlow platform [62] for DeepSeek-series model access. All experiments were on a server running Ubuntu 22.04 with an Intel Core i9-13900K processor and 128GB DRAM. Each LLM experiment was repeated 10 times to mitigate randomness.

*Baselines.* We used a diverse set of state-of-the-art smart contract vulnerability detection tools as baselines, representing different technical approaches: static analyzers (Slither [23] v0.10.4), fuzzers (Ityfuzz [61] commit e0fa1ab), symbolic execution-based tools (Mythril [12] v0.24.8), and

Table 4. Comparison results of Synapse with state-of-the-art tools.

| | Incidents | | | DeFiHacks | | | Top-100 |
|---|---|---|---|---|---|---|---|
| | Recall | Precision | F1 | Recall | Precision | F1 | FPR |
| Slither | 0.410 | 0.099 | 0.160 | 0.412 | 0.184 | 0.255 | 0.420 |
| Ityfuzz | 0.225±0.033 | 0.629±0.051 | 0.331±0.038 | 0.176±0.028 | 0.429±0.046 | 0.250±0.032 | **0.020**±0.008 |
| Mythril | 0.421±0.019 | 0.200±0.024 | 0.269±0.018 | 0.294±0.022 | 0.192±0.018 | 0.232±0.016 | 0.260±0.021 |
| GPTScan | 0.108±0.018 | 0.458±0.032 | 0.175±0.021 | 0.235±0.039 | 0.333±0.042 | 0.276±0.035 | 0.090±0.012 |
| **Synapse** | **0.719**±0.025 | **0.743**±0.021 | **0.731**±0.019 | **0.765**±0.041 | **0.542**±0.045 | **0.634**±0.036 | 0.070±0.015 |

All values represent arithmetic means over 10 independent runs. ± denotes standard deviation. Slither is deterministic and produces identical results across runs.

machine learning-based tools (GPTScan [68] commit e2d8162). Mythril and Ityfuzz ran with a 2-hour timeout, repeated 10 times to mitigate randomness.

*Metrics.* We used these metrics for performance evaluation: **TP** (true positives), **FP** (false positives), **FN** (false negatives), **TN** (true negatives), **Recall** $R = TP/(TP + FN)$, **Precision** $P = TP/(TP + FP)$, **F1** $= 2 \times P \times R/(P + R)$, and **FPR** $= FP/(FP + TN)$.

**Research Questions.** We conducted extensive experiments and analysis to answer the following research questions (RQs) concerning Synapse's effectiveness in detecting smart contract vulnerabilities.

Comparing state-of-the-art tools' vulnerability detection capabilities with our approach demonstrates Synapse's effectiveness. This comparison will highlight Synapse's strengths in identifying complex vulnerabilities missed by traditional static analyzers and fuzzers.

> **RQ1**: How effective is Synapse at detecting vulnerabilities compared to state-of-the-art tools?

Since we address several key challenges in vulnerability detection with our proposed components, we investigate:

> **RQ2**: To what extent does each key component of Synapse improve detection accuracy and capability?

Synapse's practical performance (detection accuracy and financial costs) in real-world vulnerability detection further demonstrates our approach's effectiveness, leading to:

> **RQ3**: How does Synapse perform in real-world vulnerability detection scenarios regarding accuracy and cost-effectiveness?

### 5.1 Comparative Evaluation with State-of-the-Art Tools (RQ1)

We applied Synapse to our datasets and compared results with state-of-the-art tools (baselines). Table 4 presents the comparative results. Synapse demonstrates superior performance over baseline tools, achieving a recall of 0.719. Our analysis reveals that traditional tools effectively detect machine-auditable bugs but exhibit significant limitations with machine-unauditable bugs. Symbolic execution tools like Mythril and fuzzing-based approaches such as Ityfuzz fail to identify most logic flaws due to their lack of semantic understanding of the code and reliance on predefined vulnerability detection patterns. GPTScan, while supporting 10 specific vulnerability sub-types, requires compilation and extensive manual configuration of static analysis rules, making it unable to analyze contracts with compilation failures and limiting detection to predefined patterns.

In contrast, Synapse leverages comprehensive vulnerability reasoning buffer extracted from expert auditing processes and security reports, resulting in a 6.7× improvement against GPTScan in recall when applied to real-world security incidents.

Table 5. Comparative analysis of SYNAPSE variants and zero-shot LLM performance metrics.

| Model | TP | FP | FN | TN | Recall | Precision | F1 |
|---|---|---|---|---|---|---|---|
| SYNAPSE$_b$ | 137 | 89 | 159 | 64 | 0.462 | 0.606 | 0.525 |
| SYNAPSE$_f$ | 151 | 169 | 135 | 51 | 0.528 | 0.472 | 0.498 |
| SYNAPSE$_t$ | 116 | 148 | 150 | 75 | 0.436 | 0.439 | 0.437 |
| $LLM_{r1}$ | 72 | 135 | 194 | 49 | 0.271 | 0.348 | 0.305 |
| $LLM_{o3}$ | 63 | 193 | 203 | 55 | 0.237 | 0.246 | 0.241 |
| $LLM_{claude}$ | 79 | 163 | 187 | 45 | 0.297 | 0.326 | 0.311 |
| **SYNAPSE** | 192 | 73 | 74 | 79 | **0.725** | **0.725** | **0.723** |

All values represent arithmetic means over 10 independent runs.

Table 6. Time/cost analysis and LLM sensitivity on the *Incidents* dataset. R = reasoning model; C = code model.

| Dataset | KLoC | Time (s) | | Cost ($) | |
|---|---|---|---|---|---|
| | | /KLoC | Total | /KLoC | Total |
| *Incidents* | 79.8 | 38.9 | 3100 | 0.044 | 3.5 |
| *DeFiHacks* | 19.1 | 62.7 | 1200 | 0.057 | 1.1 |
| **Average** | 49.5 | 50.8 | 2150 | 0.051 | 2.3 |

(a) Time and cost analysis

| R-Model | C-Model | Rec. | Prec. | F1 |
|---|---|---|---|---|
| R1-Distill-32B | DeepSeek-V3 | 0.719 | **0.743** | **0.731** |
| DeepSeek-R1 | DeepSeek-V3 | **0.724** | 0.738 | **0.731** |
| o3-mini | DeepSeek-V3 | 0.701 | 0.725 | 0.713 |
| Claude Sonnet | DeepSeek-V3 | 0.683 | 0.714 | 0.698 |
| R1-Distill-32B | GPT-4o | 0.708 | 0.731 | 0.719 |
| R1-Distill-32B | Claude Sonnet | 0.695 | 0.722 | 0.708 |
| R1-Distill-32B | Qwen2.5-32B | 0.689 | 0.715 | 0.702 |

(b) LLM sensitivity

For the benign dataset *Top-100*, SYNAPSE achieves a false positive rate of 7%, which is acceptable for real-world vulnerability detection scenarios. These false positives primarily stem from LLM hallucinations that occur when the model fails to follow the prompt and produces unreliable results.

We further conducted statistical significance testing to validate our results. We employed the Wilcoxon signed-rank test to compare SYNAPSE against each baseline across the 10 independent runs on the *Incidents* dataset. The results demonstrate statistically significant improvements over all baselines ($p < 0.05$). Additionally, we computed Cliff's delta effect size, which indicates **large** effect sizes ($|d| = 1.0$) for all pairwise comparisons. These results confirm that the observed performance differences are not attributable to random variation.

**Answer to RQ1:** SYNAPSE demonstrates superior detection capabilities, identifying more than **71.9%** of vulnerabilities in real-world datasets. SYNAPSE's recall is a **3**× improvement over state-of-the-art LLM-driven tools and **2**× over traditional tools on average. In summary, SYNAPSE effectively identifies logic flaws undetected by existing state-of-the-art works.

### 5.2 Effectiveness of Key Components in SYNAPSE (RQ2)

We analyzed the effectiveness of key components (Buffer of Vulnerability Reasoning Thoughts, Focal Context, Semantic Tool Layers) in SYNAPSE via ablation study. We created SYNAPSE variants by disabling BoVRT (SYNAPSE$_b$), replacing focal context with fixed $f_f$ level context (SYNAPSE$_f$), and disabling the semantic tools layer (SYNAPSE$_t$). We further evaluated SYNAPSE against state-of-the-art LLMs using zero-shot prompting: DeepSeek-R1 [28] ($LLM_{r1}$), Claude-4.0-Sonnet [1] ($LLM_{claude}$) and o3-mini [51] ($LLM_{o3}$).

Table 5 shows each component's significant contribution in SYNAPSE. Disabling BoVRT (SYNAPSE$_b$) dropped recall from 0.719 to 0.462, indicating the crucial role of vulnerability reasoning thoughts. Replacing focal context with fixed-level context (SYNAPSE$_f$) reduced recall to 0.528, highlighting adaptive context selection's importance. Disabling the semantic tools layer (SYNAPSE$_t$) made short-term context significantly large and inefficiently utilized vulnerability reasoning knowledge, resulting in 0.436 recall.

Table 7. Focal context granularity and Verifier effectiveness on the *Incidents* dataset.

| Granularity | Rec. | Prec. | F1 |
|---|---|---|---|
| $f_f$ only | 0.578 | 0.472 | 0.520 |
| $f_f$+$f_m$ | 0.621 | 0.538 | 0.577 |
| $f_f$+$f_m$+$f_i$ | 0.674 | 0.651 | 0.662 |
| $f_f$+$f_m$+$f_i$+$f_v$ | 0.697 | 0.706 | 0.701 |
| Agent-selected | **0.719** | **0.743** | **0.731** |

(a) Focal context granularity

| Metric | Result |
|---|---|
| TP correctly passed | 95.2% |
| FP correctly filtered | 91.9% |
| TP incorrectly rejected | 4.8% |
| Recall w/o Verifier | 0.755 |
| Prec. w/o Verifier | 0.612 |
| Recall w/ Verifier | **0.719** |
| Prec. w/ Verifier | **0.743** |

(b) Verifier effectiveness

Moreover, comparative assessments with zero-shot LLMs demonstrate that even advanced models such as DeepSeek-R1 and Claude-4.0-Sonnet attain significantly inferior F1 scores (below 0.297) in contrast to Synapse. Our analysis elucidates that although these models are capable of independently producing elaborate Chains-of-Thought (CoT), their comprehension of smart contract vulnerabilities remains deficient. Furthermore, the single-pass inference methodology employed by LLMs is susceptible to factuality and faithfulness hallucinations [29], leading to erroneous detections.

Regarding data contamination, less than **3.9%** of evaluation functions overlap with the BoVRT corpus at the function level. Moreover, the *Incidents* dataset comprises vulnerabilities discovered in 2024, with the majority postdating the training data cutoff of our backend LLMs. The poor zero-shot recall of these LLMs (below 0.297) further suggests limited pre-existing knowledge of evaluation contracts. The LLM sensitivity analysis (Section 5.2.1) corroborates this, as Synapse maintains consistent performance across models with different training cutoff dates.

Our components mitigate LLMs' domain knowledge limitations and critical code identification challenges, yielding a 2.5× recall improvement.

*5.2.1 LLM Sensitivity Analysis.* Table 6 presents Synapse's performance across different reasoning and code model combinations on the *Incidents* dataset. Synapse maintains stable performance across all combinations, with recall ranging from 0.683 to 0.724. Notably, varying the reasoning model produces larger performance differences (recall from 0.683 to 0.724) than varying the code model (from 0.689 to 0.719), suggesting reasoning capability matters more. The default configuration (DeepSeek-R1-Distill-32B + DeepSeek-V3) achieves a favorable cost-performance balance, as the full DeepSeek-R1 model yields only marginal improvement at substantially higher cost.

*5.2.2 Focal Context Granularity Analysis.* Table 7(a) compares fixed granularity configurations against Synapse's agent-selected adaptive approach. Performance improves progressively with additional context levels, with the largest gain from adding internal calls ($f_i$), which improves F1 by 0.085. The agent-selected approach outperforms even the full fixed configuration by 0.030 in F1, indicating that adaptive granularity selection effectively balances context richness against noise.

*5.2.3 Verifier Agent Effectiveness.* Table 7(b) presents the Verifier agent's effectiveness. The Verifier correctly filters 57 false positives while incorrectly rejecting only 9 true positives (4.8% false rejection rate). Without the Verifier, precision drops from 0.743 to 0.612, confirming its critical role in maintaining detection reliability.

**Answer to RQ2:** Our proposed components (BoVRT, Focal Context, Semantic Tools) effectively address LLMs' inherent limitations, yielding a 2.5× recall improvement. The framework demonstrates stable performance across different LLM combinations, and the adaptive focal context and **Verifier** agent each contribute meaningfully to detection capability.

Table 8. Statistics of vulnerabilities detected by SYNAPSE in contracts on BSC (Feb-Mar 2025).

| Vulnerability Type | Severity | Number | Baseline |
|---|---|---|---|
| Access Control | High | 56 | 5 |
| Price Manipulation | High | 6 | ✗ |
| Invariants Violation | High | 2 | ✗ |
| LP Token Burn | High | 2 | ✗ |
| Initialization | High | 48 | 3 |
| Authorization Failure | High | 2 | ✗ |
| Privilege Escalation | High | 1 | ✗ |
| **Total** | | 117 | 8 |

```
function _airdrop(address from, address to, uint256 tAmount) private {
    uint256 num = _airdropLen;
    uint256 seed = (uint160(lastAirdropAddress) | block.number) ^ (uint160(from) ^ uint160(to));
    uint256 airdropAmount = _airdropAmount;
    address airdropAddress;
    for (uint256 i; i < num;) {
        airdropAddress = address(uint160(seed | tAmount)); // vulnerable: predictable seed, manipulable address
        _balances[airdropAddress] = airdropAmount;
        emit Transfer(airdropAddress, airdropAddress, airdropAmount);
        unchecked{
            ++i;
            seed = seed >> 1;
        }
    }
    lastAirdropAddress = airdropAddress;
}
```

Fig. 7. Arbitrary balance manipulation vulnerability: predictable seed in airdrop function (internally called by `transfer`) allows LP balance manipulation.

```
#[storage(read)]
fn is_borrow_collateralized(account: Identity) -> bool {
    // ... code omitted for brevity
    while index < len {
        // ... code omitted for brevity
        let price = get_price_internal(collateral_configuration.price_feed_id);
        // missing price.confidence check
        let price = u256::from(price.price);
        let amount = balance * price / price_scale;
        borrow_limit += amount * collateral_configuration.borrow_collateral_factor * base_scale / FACTOR_SCALE_18 / collateral_scale;
        index += 1;
    }
}
```

Fig. 8. Price oracle validation vulnerability in a Sway-based lending protocol. The function fails to validate the confidence level of the price feed, allowing attackers to potentially manipulate collateral values during periods of high price volatility.

### 5.3 Real-world Performance (RQ3)

We evaluated detection time and financial cost for *Incidents* and *DeFiHacks* datasets. Table 6 shows time and financial costs for evaluating these datasets across model variants. Financial cost was calculated by API usage credit differences before and after evaluation.

Average processing time per thousand lines of code (KLoC) across datasets was 50.8 seconds, with a corresponding $0.051/KLoC financial cost. For entire datasets, detection took approximately 2,150 seconds (35.8 minutes) and $2.3 in API costs. The *Incidents* dataset had lower processing time per KLoC (38.9s) than *DeFiHacks* (62.7s), attributable to our preprocessor excluding standard implementations, meaning fewer functions needed SYNAPSE analysis in the *Incidents* dataset.

To assess the real-world efficacy of Synapse, we conducted an in-depth analysis of smart contracts deployed on the Binance Smart Chain (BSC) between February and March 2025. We gathered contracts by indexing contract creation transactions from block 46262222 to 47961273. We refined this dataset by implementing transaction activity filters, requiring at least one transaction to the contract address, and applying heuristic rules to exclude proxy and other irrelevant implementations. This process resulted in a curated set of 25,000 contracts for vulnerability detection. Notably, we employed Heimdall [2] to decompile closed-source contracts, enabling Synapse to analyze them through its adaptable AST parsing in the preprocessor. Detection outcomes were manually verified for accuracy. The results are presented in Table 8. Upon re-evaluating the identified vulnerable contracts with baseline tools, only 8 out of 117 vulnerabilities were detected. As shown in Table 8, the majority of identified vulnerabilities pertain to access control and contract initialization issues.

Figure 7 presents a critical vulnerability. The private function `_airdrop`, invoked internally by `transfer`, relies on `tAmount` derived from the transfer amount. Due to a predictable seed, an attacker can craft a specific token transfer amount to manipulate `airdropAddress` to match the liquidity provider's (LP) address. As a result, upon invoking `transfer`, the attacker can overwrite the LP's token balance to a minimal airdrop amount, thereby distorting the LP's token price.

*Case Study: Critical Vulnerability Affecting $30 Million in Assets.* To explore Synapse's capability beyond Solidity, we analyzed Sway contracts on the Fuel blockchain. As shown in Figure 8, Synapse identified a critical price oracle validation vulnerability in a Sway-based lending protocol with total value locked (TVL) exceeding **$30 million**. While `get_price_internal` checks the Pyth price feed confidence, this validation is not incorporated into collateral calculations, which use raw prices without accounting for uncertainty. According to the Pyth oracle documentation, conservative collateral pricing should employ `price - confidence` to protect against price volatility. Synapse detected this vulnerability through a *targeted thought* template for price oracle manipulation, which guided the **Researcher** agents to examine the price feed consumption path and verify whether confidence intervals were properly incorporated. Following responsible disclosure, the development team acknowledged and patched the vulnerability.

**Answer to RQ3:** Synapse is highly cost-efficient, requiring less than $5 for vulnerability detection across our datasets, with per-KLoC cost less than $0.05. This is a substantial economic advantage over human security audits (typically starting at $1,000 for basic projects). Additionally, Synapse identified 117 previously undiscovered vulnerabilities missed by baseline tools, highlighting its superior detection.

*Responsible Disclosure.* We initiated responsible disclosure for the 117 identified vulnerabilities. For contracts with identifiable teams or bug bounty programs, we submitted reports through appropriate channels. All acknowledged vulnerabilities have been patched by the respective development teams as of the submission date.

### 5.4 Error Analysis

We systematically analyzed false positives and false negatives on the *Incidents* dataset. Table 9 presents the distribution. False positives stem from four causes: (1) *factuality hallucinations*, where the LLM generates claims contradicting the code; (2) *faithfulness hallucinations*, where the LLM introduces assumptions beyond the provided context; (3) *insufficient context*, where the focal context excludes code critical to the assessment; and (4) *template mismatch*, where the retrieved thought does not match the contract's semantics. False negatives arise from: (1) *no matching template* in the BoVRT corpus; (2) *semantic search failure* for functions with non-descriptive names; (3) *focal*

Table 9. Distribution of error causes on the *Incidents* dataset (62 FPs, 70 FNs).

| FP Cause | # |
|---|---|
| Factuality Halluc. | 24 (38.7%) |
| Faithfulness Halluc. | 17 (27.4%) |
| Insufficient Context | 13 (21.0%) |
| Template Mismatch | 8 (12.9%) |

(a) False positives

| FN Cause | # |
|---|---|
| No Matching Template | 28 (40.0%) |
| Semantic Search Fail. | 18 (25.7%) |
| Focal Context Excl. | 15 (21.4%) |
| Verifier Rejection | 9 (12.9%) |

(b) False negatives

*context exclusion* of modified standard library functions; and (4) *incorrect Verifier rejection* due to minor imprecisions in proof-of-thoughts.

We illustrate the two most prevalent error categories with representative examples.

**False positive example (factuality hallucination):** In a lending protocol, Synapse reported that the `liquidate` function lacked a health factor check before seizing collateral. However, the health factor validation was present in an internal function `_validateLiquidation` invoked by `liquidate`. The LLM incorrectly asserted the absence of this check despite having the function body in its focal context, generating a factually inaccurate claim.

**False negative example (no matching template):** A yield aggregator contract contained a vulnerability where the `harvest` function did not verify the legitimacy of the yield source, allowing an attacker to inject a malicious strategy contract. This vulnerability pattern (strategy injection via unvalidated yield sources) was not represented in the BoVRT corpus, as no similar exploit existed in the collected audit reports. Consequently, no relevant thought template was retrieved, and the *Researcher* agents did not examine this attack vector.

## 6 Discussion

In this section, we discuss the current limitations of Synapse.

*Limitations Due to the Backend LLMs.* The performance of Synapse is closely tied to the capabilities of the backend LLMs. Fine-tuning or setting up local LLMs requires significant computing resources and is costly, making it infeasible for smaller teams. For example, fine-tuning the GPT-4o model on our collected audit reports costs at least $500 for ten rounds of batches. Consequently, we utilize API-based inference for LLMs, which costs less than $0.1 per thousand lines of code. Our current model selection balances capabilities and cost across various models (e.g., Claude-series, Gemini, GPT-series, DeepSeek-series, DouBao-series, Qwen-series). Additionally, despite our efforts to mitigate hallucinations in LLMs through focal context and the buffer of thoughts, these issues may still occur. Factuality and faithfulness hallucinations [29] remain an inherent challenge in current LLM technology, potentially leading to inaccurate results during vulnerability reasoning. As LLM technology rapidly evolves, we plan to explore both newer models with improved reasoning capabilities and cost-effective local LLM inference options in future work to further improve detection accuracy.

*Limitations in the Vulnerability Detection Types.* Although Synapse outperforms state-of-the-art vulnerability detection tools in detecting logic flaws, which constitute more than 80% of exploitable bugs, it is not specifically designed for detecting Machine-Auditable Bugs (MABs). The vulnerability reasoning approach Synapse leverages is distilled from audit reports that primarily focus on logic flaws. Hence, these reasoning thoughts are not designed for detecting MABs. Traditional static analysis tools are more suitable for detecting MABs as they produce results within seconds and are more cost-effective for large-scale analysis. We aim to shed light on the potential of enhancing LLMs in detecting logic flaws which are exploitable in real-world scenarios.

## 7 Related Work

*Traditional Analysis Tools.* Traditional approaches for smart contract vulnerability detection encompass *static analysis* [7, 36, 72, 85], where tools such as Slither [23], Ethainter [6], and Securify [75] detect vulnerabilities under predefined rules without execution; *symbolic execution* [32], where Mythril [12] and Manticore [43] explore execution paths with symbolic values; and *fuzzing* [9, 31], where Ityfuzz [61], Confuzzius [73], and sfuzz [48] identify vulnerabilities through dynamic oracle violation detection. SmartOracle [65] generates invariants from transaction history but is inherently limited for newly deployed contracts lacking historical data. Sejfia et al. [56] highlighted that multi-base unit (MBU) vulnerabilities, which span multiple contracts or code units, expose significant limitations in existing detection tools that analyze contracts in isolation. While these approaches produce reliable results for machine-auditable vulnerabilities, Zhang et al. [89] indicates that more than 80% of exploitable bugs are logic flaws that these tools fail to detect. *Unlike the above works, Synapse leverages focal context and cross-contract semantic search to reason across multiple code units, addressing both MBU vulnerabilities and machine-unauditable logic flaws.*

*LLM-based Analysis Tools.* Large language models (LLMs) have demonstrated significant efficacy in various code analysis tasks. Recent research [67, 87] has systematically investigated the capabilities of general-purpose LLMs in smart contract vulnerability detection. GPTScan [68] implements a binary classification approach where LLMs determine the presence of predefined vulnerabilities. Ma et al. [40] proposed a hybrid methodology that integrates fine-tuned models with agents to provide justified smart contract audits. SmartInv [77] employs Tree of Thoughts (ToT) prompting to analyze multiple smart contract modalities and generate invariants. While these invariants can highlight potential vulnerabilities, SmartInv itself does not perform direct vulnerability detection. Chen et al. [8] conducted an evaluation comparing GPT models against traditional security tools, revealing varied effectiveness among different vulnerability types in smart contracts.

Researchers have also explored open-source models for smart contract security. LLM4Fuzz [60] employs Llama [74] to guide contract fuzzers toward high-value code regions, VulCoBERT [81] leverages fine-tuned CodeBERT representations for vulnerability detection, and Deng et al. [21] demonstrated LLMs as effective edge-case generators for fuzzing deep learning libraries. However, these approaches suffer from limited context, inadequate tooling, and manual rule definition. *Unlike previous approaches that rely on fine-tuned models [81], predefined vulnerability patterns, or historical transaction data, Synapse addresses these limitations through its combination of semantic tools, adaptive focal context, and buffer of vulnerability reasoning thoughts.*

## 8 Conclusion

In this paper, we present Synapse, an advanced smart contract detection framework leveraging thought-augmented LLM and fine-grained analysis. We utilize focal context during LLM analysis to improve reasoning accuracy under limited context windows. To mimic security researchers' behavior, we design semantic tools for multi-agents during their reasoning-acting process. To address limited vulnerability knowledge, we propose a buffer of vulnerability reasoning thoughts to enhance detection capabilities. Synapse outperforms state-of-the-art vulnerability detection tools in detecting logic flaws in terms of recall under real-world datasets. Furthermore, Synapse has identified 117 newly discovered vulnerabilities in on-chain smart contracts. Notably, the detection of one critical vulnerability safeguarded assets totaling $30 million from potential losses.

## Data Availability

We publish the source code of Synapse at https://zenodo.org/records/17127186.

## Acknowledgments

We thank all anonymous reviewers for their invaluable comments.

## References


[1] Anthropic. 2024. Claude: Conversational AI for Productivity. https://claude.ai/.

[2] Jon Becker. 2025. Heimdall: An advanced EVM smart contract toolkit for forensic and heuristic analysis. https://github.com/Jon-Becker/heimdall-rs.

[3] Aleksander Berentsen and Fabian Schär. 2019. Stablecoins: The quest for a low-volatility cryptocurrency. *The economics of Fintech and digital currencies* (2019), 65–75.

[4] BlockSec. 2025. BlockSec: Full-Stack Blockchain Security Service Provider. https://blocksec.com.

[5] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 161–178.

[6] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 454–469.

[7] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981* (2018).

[8] Chong Chen, Jianzhong Su, Jiachi Chen, Yanlin Wang, Tingting Bi, Jianxing Yu, Yanli Wang, Xingwei Lin, Ting Chen, and Zibin Zheng. 2023. When chatgpt meets smart contract vulnerability detection: How far are we? *ACM Transactions on Software Engineering and Methodology* (2023).

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

[10] Chroma. 2023. Chroma: the open-source embedding database. https://github.com/chroma-core/chroma.

[11] Codeium. 2024. Windsurf: Built to keep you in flow state. https://codeium.com/windsurf.

[12] ConsenSysDiligence. 2025. Mythril: A symbolic-execution-based security analysis tool for EVM bytecode. https://github.com/ConsenSysDiligence/mythril.

[13] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, et al. 2023. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062* (2023).

[14] Cursor. 2024. Cursor: The AI Code Editor. https://www.cursor.com.

[15] Seyed Shayan Daneshvar. 2025. Exploring representation-level augmentation and RAG-based vulnerability augmentation with LLMs for vulnerability detection. (2025).

[16] Isaac David, Liyi Zhou, Kaihua Qin, Dawn Song, Lorenzo Cavallaro, and Arthur Gervais. 2023. Do you still need a manual smart contract audit? *arXiv preprint arXiv:2306.12338* (2023).

[17] Decurity. 2025. Decurity: Decentralized Finance Systems Application Security. https://www.decurity.io.

[18] DeepSeek AI. 2024. DeepSeek-R1-Distill-Qwen-32B: A distilled reasoning model based on Qwen architecture. https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Qwen-32B.

[19] DeepSeek AI. 2024. DeepSeek-V3: A collection of large language models. https://huggingface.co/deepseek-ai/DeepSeek-V3.

[20] DefiLlama. 2025. DefiLlama Hacks: DeFi Exploits and Hacks Database. https://defillama.com/hacks. Accessed: 2025.

[21] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[22] Yuying Du and Xueyan Tang. 2024. Evaluation of ChatGPT's Smart Contract Auditing Capabilities Based on Chain of Thought. *arXiv preprint arXiv:2402.12023* (2024).

[23] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.

[24] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[25] Foundry. 2023. Foundry: A blazing fast, portable and modular toolkit for Ethereum application development. https://github.com/foundry-rs/foundry.

[26] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* 2 (2023).

[27] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and*

*analysis*. 557–560.

[28] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[29] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2025. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems* 43, 2 (2025), 1–55.

[30] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-Coder Technical Report. *arXiv preprint arXiv:2409.12186* (2024).

[31] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 259–269.

[32] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[33] Yury Kuratov, Aydar Bulatov, Petr Anokhin, Ivan Rodkin, Dmitry Sorokin, Artyom Sorokin, and Mikhail Burtsev. 2024. Babilong: Testing the limits of llms with long context reasoning-in-a-haystack. *Advances in Neural Information Processing Systems* 37 (2024), 106519–106554.

[34] LangChain. 2023. LangChain: A framework for building LLM-powered applications. https://github.com/langchain-ai/langchain.

[35] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[36] Zeqin Liao, Zibin Zheng, Xiao Chen, and Yuhong Nan. 2022. SmartDagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 752–764.

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

[38] Lu Liu, Sicong Zhou, Huawei Huang, and Zibin Zheng. 2021. From technology to society: An overview of blockchain-based DAO. *IEEE Open Journal of the Computer Society* 2 (2021), 204–215.

[39] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173.

[40] Wei Ma, Daoyuan Wu, Yuqiang Sun, Tianwen Wang, Shangqing Liu, Jian Zhang, Yue Xue, and Yang Liu. 2024. Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications. *arXiv preprint arXiv:2403.16073* (2024).

[41] Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. 2023. Prompt engineering in large language models. In *International conference on data intelligence and cognitive informatics*. Springer, 387–402.

[42] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* 26 (2013).

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

[44] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[45] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. 2023. A comprehensive overview of large language models. *arXiv preprint arXiv:2307.06435* (2023).

[46] Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. 2005. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*. 1–5.

[47] Hoang H Nguyen, Nhat-Minh Nguyen, Hong-Phuc Doan, Zahra Ahmadi, Thanh-Nam Doan, and Lingxiao Jiang. 2022. Mando-guru: Vulnerability detection for smart contract source code by heterogeneous graph embeddings. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1736–1740.

[48] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 778–788.

[49] Nomic Foundation. 2023. Hardhat: An Ethereum development environment for professionals. https://github.com/NomicFoundation/hardhat.

[50] OpenAI. 2022. ChatGPT. https://openai.com/index/chatgpt. Accessed: 2024.

[51] OpenAI. 2025. OpenAI o3-mini. https://platform.openai.com/docs/models/o3-mini.

[52] OpenAI. 2025. OpenAI Platform: API for Accessing GPT Models. https://platform.openai.com.

[53] Terence J. Parr and Russell W. Quong. 1995. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810.

[54] Qwen Team. 2024. Qwen2.5-32B. https://huggingface.co/Qwen/Qwen2.5-32B.

[55] Michael Rodler, David Paaßen, Wenting Li, Lukas Bernhard, Thorsten Holz, Ghassan Karame, and Lucas Davi. 2023. EF\CF: High Performance Smart Contract Fuzzing for Exploit Generation. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 449–471.

[56] Adriana Sejfia, Satyaki Das, Saad Shafiq, and Nenad Medvidović. 2024. Toward Improved Deep Learning-Based Vulnerability Detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[57] Christoph Sendner, Huili Chen, Hossein Fereidooni, Lukas Petzi, Jan König, Jasper Stang, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. 2023. Smarter Contracts: Detecting Vulnerabilities in Smart Contracts with Deep Transfer Learning.. In *NDSS*.

[58] Christoph Sendner, Lukas Petzi, Jasper Stang, and Alexandra Dmitrienko. 2024. Large-scale study of vulnerability scanners for Ethereum smart contracts. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2273–2290.

[59] Samiha Shimmi, Yash Saini, Mark Schaefer, Hamed Okhravi, and Mona Rahimi. 2024. Software Vulnerability Detection Using LLM: Does Additional Information Help?. In *2024 Annual Computer Security Applications Conference Workshops (ACSAC Workshops)*. IEEE, 216–223.

[60] Chaofan Shou, Jing Liu, Doudou Lu, and Koushik Sen. 2024. Llm4fuzz: Guided fuzzing of smart contracts with large language models. *arXiv preprint arXiv:2401.11108* (2024).

[61] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. Ityfuzz: Snapshot-based fuzzer for smart contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 322–333.

[62] SiliconFlow. 2025. SiliconFlow: High-Performance AI Computing Platform. https://siliconflow.cn.

[63] SlowMist. 2024. Analysis of the UwU Lend Hack. https://slowmist.medium.com/analysis-of-the-uwu-lend-hack-9502b2c06dbe. Accessed: 2024.

[64] James Stanier and Des Watson. 2013. Intermediate representations in imperative compilers: A survey. *ACM Computing Surveys (CSUR)* 45, 3 (2013), 1–27.

[65] Jianzhong Su, Jiachi Chen, Zhiyuan Fang, Xingwei Lin, Yutian Tang, and Zibin Zheng. 2025. SmartOracle: Generating Smart Contract Oracle via Fine-Grained Invariant Detection. *IEEE Transactions on Software Engineering* (2025).

[66] Dianxiang Sun, Wei Ma, Liming Nie, and Yang Liu. 2024. Sok: Comprehensive analysis of rug pull causes, datasets, and detection tools in defi. *arXiv preprint arXiv:2403.16082* (2024).

[67] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Yang Liu, and Yingjiu Li. 2024. Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning. *arXiv preprint arXiv:2401.16185* (2024).

[68] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[69] LMArena Team. [n. d.]. Chatbot Arena LLM Leaderboard. https://lmarena.ai.

[70] Munchables Team. 2024. Munchables: A GameFi Project on Blast. https://github.com/code-423n4/2024-07-munchables.

[71] TenArmor. 2025. TenArmor: Blockchain Security Solutions. https://tenarmor.com.

[72] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*. 9–16.

[73] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 103–119.

[74] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[75] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 67–82.

[76] Velocore. 2024. Velocore Incident Post-Mortem. https://velocorexyz.medium.com/velocore-incident-post-mortem-6197020ec3e9. Accessed: 2024.

[77] Sally Junsong Wang, Kexin Pei, and Junfeng Yang. 2024. Smartinv: Multimodal learning for smart contract invariant inference. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2217–2235.

[78] Weizhi Wang, Li Dong, Hao Cheng, Xiaodong Liu, Xifeng Yan, Jianfeng Gao, and Furu Wei. 2023. Augmenting language models with long-term memory. *Advances in Neural Information Processing Systems* 36 (2023), 74530–74543.
[79] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171* (2022).
[80] Zhiyuan Wei, Jing Sun, Zijiang Zhang, Xianhao Zhang, Meng Li, and Zhe Hou. 2024. LLM-SmartAudit: Advanced Smart Contract Vulnerability Detection. *arXiv preprint arXiv:2410.09381* (2024).
[81] Yuying Xia, Haijian Shao, and Xing Deng. 2024. VulCoBERT: A CodeBERT-based System for Source Code Vulnerability Detection. In *Proceedings of the 2024 International Conference on Generative Artificial Intelligence and Information Security*. 249–252.
[82] Jiahua Xu and Yebo Feng. 2022. Reap the harvest on blockchain: A survey of yield farming protocols. *IEEE Transactions on Network and Service Management* 20, 1 (2022), 858–869.
[83] Jiahua Xu, Krzysztof Paruch, Simon Cousaert, and Yebo Feng. 2023. Sok: Decentralized exchanges (dex) with automated market maker (amm) protocols. *Comput. Surveys* 55, 11 (2023), 1–50.
[84] Jiahua Xu and Nikhil Vadgama. 2022. From banks to defi: the evolution of the lending market. *Enabling the Internet of Value: How Blockchain Connects Global Businesses* (2022), 53–66.
[85] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. 2020. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1029–1040.
[86] Ling Yang, Zhaochen Yu, Tianjun Zhang, Shiyi Cao, Minkai Xu, Wentao Zhang, Joseph E Gonzalez, and Bin Cui. 2024. Buffer of thoughts: Thought-augmented reasoning with large language models. *Advances in Neural Information Processing Systems* 37 (2024), 113519–113544.
[87] Jeffy Yu. 2024. Retrieval Augmented Generation Integrated Large Language Models in Smart Contract Vulnerability Detection. *arXiv preprint arXiv:2407.14838* (2024).
[88] Wuqi Zhang, Zhuo Zhang, Qingkai Shi, Lu Liu, Lili Wei, Yepang Liu, Xiangyu Zhang, and Shing-Chi Cheung. 2024. Nyx: Detecting exploitable front-running vulnerabilities in smart contracts. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2198–2216.
[89] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying exploitable bugs in smart contracts. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 615–627.
[90] Arastoo Zibaeirad and Marco Vieira. 2025. Reasoning with LLMs for Zero-Shot Vulnerability Detection. *arXiv preprint arXiv:2503.17885* (2025).

Received 2026-02-23; accepted 2026-03-24