NASC: Neuro-symbolic Architecture for Smart Contracts Vulnerability Detection

Cleidimar L. dos Passos
Universidade Federal de Viçosa
Florestal, Brasil
cleidimar.passos@ufv.br

Josué N. Campos
Universidade Federal de Viçosa
Florestal, Brasil
josue.campos@ufv.br

Luís H. S. de Carvalho
Universidade Federal de Viçosa
Florestal, Brasil
luis.h.carvalho@ufv.br

Matheus J. da Silva
Universidade Federal de Viçosa
Florestal, Brasil
matheus.junio@ufv.br

José A. M. Nacif
Universidade Federal de Viçosa
Florestal, Brasil
jnacif@ufv.br

Abstract

Smart contracts control digital assets under conditions of immutability in blockchain networks, making vulnerability detection an important task in decentralized systems security. Traditional static analysis tools generally rely on specific syntactic patterns, which can make it difficult for developers to interpret the results. This work presents NASC, a Neuro-symbolic agent architecture for vulnerability analysis in Solidity smart contracts. The approach combines a language model-based planner agent, a symbolic core responsible for converting the source code into first-order logic facts, and an LLM agent orchestration module based on the BMAD (Build More Architect Dreams) method that enables the agent to read, write, and run the symbolic core of NASC. The detection mechanism is performed by declarative rules implemented in Prolog, while an analyst agent transforms the results into structured natural language reports. The architecture was evaluated using three literature datasets. Our results show that NASC is capable of identifying different vulnerability patterns and producing interpretable reports, achieving over 80% of recall in 2 of 3 datasets analyzed. Furthermore, the study highlights the potential of integrating logical inference and language model-based agents to support security analysis processes.

Keywords

Neuro-symbolic AI, Smart Contracts, Vulnerability Detection

1 Introduction

Smart contracts serve as the foundational infrastructure managing hundreds of billions of dollars in decentralized finance (DeFi). However, this concentration of capital has triggered an unprecedented surge in malicious exploits, with cumulative ecosystem losses from code-level breaches surpassing several billion dollars [11]. A single vulnerability in contract logic can inflict catastrophic financial ruin in seconds.

Because these contracts are immutable once deployed, a single vulnerability in their business logic can result in catastrophic financial losses [15]. To mitigate these risks, automated vulnerability detection tools (*e.g.*, static and dynamic analyzers) serve as the industry's standard to automate the security audit process. However, traditional tools often present a severe comprehension barrier. They produce low-level logs, high false-positive rates, and abstract warnings that require significant time and manual expert analysis [3]. Consequently, professional smart contract audits remain highly expensive and can take weeks to complete, creating a bottleneck for secure software development.

Recently, Large Language Models (LLMs) have emerged as a disruptive alternative for automated vulnerability detection due to their deep understanding of natural language and code semantics [7]. Despite their promise, LLMs suffer from an inherent probabilistic nature, leading to non-deterministic outputs, code hallucinations, and an inability to provide verifiable proofs for their findings. Extracting the precise underlying reasoning behind an LLM's detection remains a notorious challenge. This leaves security engineers with a difficult choice: rely on rigid, hard-to-interpret symbolic tools, or utilize fluid, but fundamentally untrustworthy, language models. To overcome these limitations, Neuro-symbolic AI has emerged as a powerful paradigm that combines the data-driven semantic capabilities of neural networks with the rule-based rigor of symbolic logic in order to reduce the hallucination problem of LLMs [13].

To address this problem, we introduce NASC (Neuro-symbolic Architecture for Smart Contracts), a novel and decoupled architecture that fuses the deterministic nature of first-order logic with the cognitive flexibility of Multi-Agent Systems. NASC bridges the comprehension barrier by splitting the execution workload into two distinct layers: a deterministic symbolic core that guarantees precise vulnerability matching, and a neural multi-agent layer that translates these logical proofs into comprehensive, human-readable security reports. The primary contributions of this work are summarized as follows:

- We present NASC, an architecture that decouples deterministic logical inference from natural language explanation, eliminating LLM hallucinations in the detection phase while retaining human-readable reporting;
- We establish a systematic method for converting Prolog-derived vulnerability facts and Abstract Syntax Tree (AST) execution paths into structured inputs for LLM agents using The Edge Agent (TEA) and BMAD (Build More Architect Dreams) frameworks;
- We evaluate NASC against state-of-the-art datasets and contextualize with standalone static analyzers, demonstrating a significant increase in recall (80%) on these datasets.

Our work addresses automated security analysis of smart contracts, but the contribution is not just the detector. NASC automates part of the analysis workflow: symbolic detection and localization, followed by the interpretation and communication of findings via LLM agents. Therefore, the work falls under software testing by automating the security analysis activity and by automated generation of analysis artifacts.

2 Related Work

The field of automated smart contract vulnerability detection has expanded rapidly, primarily diverging into two core methodological domains: static and dynamic analysis. While these paradigms have advanced detection accuracy, they universally suffer from a comprehension barrier, presenting developers with cryptic logs, abstract warnings, or raw bytecode traces that require heavy manual verification.

Static analysis tools evaluate contract source code or compiled bytecode without executing the program. Slither [9] passes Solidity source code through an intermediate representation known as SlithIR, matching security flaws against predefined taint patterns. While highly scalable, Slither's warnings are fundamentally low-level and lack context-aware rationale. Targeting domain-specific bugs, eTainter [5] leverages inter-procedural static taint tracking within EVM bytecode to identify gas-related and Denial-of-Service vulnerabilities. Although effective within their scope, these tools remain decoupled from natural language explanation. They isolate syntactic irregularities without providing developers with reproducible or conceptually descriptive narratives explaining the vulnerability's root cause.

In order to achieve higher mathematical rigor and reduce false positives, tools like Mythril [6] use symbolic execution to explore cryptographic and operational execution paths directly on EVM bytecode. However, Mythril is bound by path explosion and generates highly abstract symbolic constraints that are conceptually distant from the original developer's code. Similarly, Sailfish [1] intercepts state-inconsistency bugs by assembling a Storage Dependency Graph (SDG) and refining potential hazards through value-summary symbolic evaluation. While Sailfish improves precision, interpreting its structural graph queries remains highly complex for developers who lack specialized knowledge in formal verification.

Dynamic testing provides concrete proof-of-concept exploits by executing contract states. Echidna [10] implements property-based fuzzing, requiring security engineers to manually formulate assertions and invariants that the tool attempts to break via randomized transaction generation. Conversely, Smartian [4] automates part of this pipeline by acting as a hybrid grey-box fuzzer, using static data-flow analysis to guide transaction sequence synthesis in an isolated EVM environment. Despite their high precision, fuzzers require active runtime setups, possess significant computational overhead, and only output minimal failing transaction counterexamples, providing zero interpretative feedback regarding why or how the underlying logic failed.

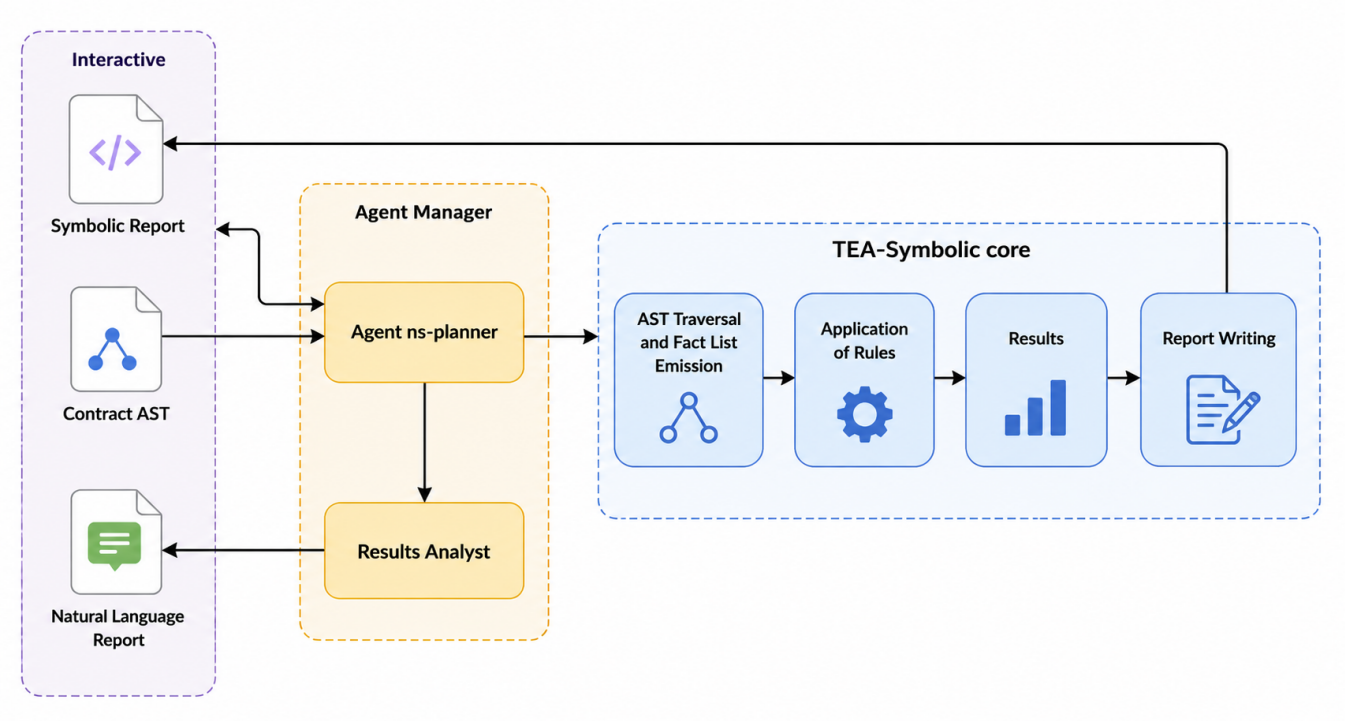

Figure 1: NASC - Agents and symbolic core.

Unlike dynamic fuzzers (Echidna, Smartian) or symbolic engines (Slither, Mythril), NASC operates completely offline without code execution or EVM emulation. By parsing the target smart contract into a structured AST, NASC maps abstract syntax nodes directly into declarative first-order logic facts. The execution of a Prolog inference engine (via the TEA framework) guarantees that vulnerability matching is deterministic, traceable, and reproducible. The core differentiator of NASC lies in its neural layer managed by the BMAD-METHOD framework. Instead of leaving the user with non-descriptive or highly abstract warnings typical of traditional analyzers, NASC parses the explicit execution paths generated by the Prolog engine, transcribing complex logical proofs into natural language security reports. Table 1 summarizes our contributions against the current literature regarding automated vulnerability detection.

Table 1: NASC contributions against the SOTA tools.

Tool	Execution-Free	Source AST Level	Deterministic Logic
Slither	✓	✓	✓
eTainter	✓	×	✓
Mythril	×	×	✓
Sailfish	✓	✓	✓
Echidna	×	✓	×
Smartian	×	×	×
NASC (Ours)	✓	✓	✓

3 Methodology

3.1 Artifacts Used and Developed

Implementation-wise, NASC combines existing components with components developed specifically for this work. TEA [2] is reused as is to run the symbolic core, BMAD [12] is also used as is. The agentic layer that runs the agents from BMAD [12], including the Agent Manager, the ns-planner and Results Analyst corresponding agent description files, were developed as part of NASC. The symbolic core file code as well as the prolog rule files were developed as part of NASC.

3.2 Architecture Overview

The diagram in Figure 1 uses a informal notation to illustrate how NASC operates. Arrows represent the data flow, while dotted boxes represent the files and components that the user can interact with separated by scope(Agent manager scope and symbolic core scope), including the Agent Manager and the agent description files it uses, as well as the TEA-Symbolic core and its symbolic pipeline.

A typical use case is as follows: the user runs the Agent Manager with the `ns-planner` agent and provides the path to the contract AST to be analyzed. The `ns-planner` agent then invokes the TEA-Symbolic core, passing the path to the contract. The TEA-Symbolic core reads the AST nodes and generates a list of facts. The rules responsible for identifying vulnerability patterns are subsequently applied to these facts, producing the analysis results, which are written to a report. This report corresponds to the *Symbolic Report* shown inside the dotted box named *Interactive* in Figure 1.

Once the Symbolic Report is generated, the `ns-planner` agent invokes the `Results Analyst` agent, passing the path to the report. The Results Analyst then reads the report and generates a *Natural Language Report* that explains the findings in natural language.

Therefore essentially the agent layer runs the TEA pipeline using command line instructions and help the user interpret the report results of the symbolic layer. The Core idea of NASC is to use agents to automatize and facilitate contract analysis and interpretation acting as a intermediate layer.

NASC enforces a strict separation of concerns, dividing the workload between the symbolic core that is executed by TEA and the agent orchestration that is the responsibility of the agent manager. NASC works by using an agent manager to orchestrate agents and interact with the user. The agents are:

- Planner
- Results Analyst

These agents run independently of the symbolic core. Their personas are defined by description files that guide them on what their responsibilities are, what they are supposed to do, as well as what is expected. In this way, the user can indicate the smart contract AST path, and the planner agent will find it. It will then execute TEA[2] to run the symbolic inference on the AST. Once the inference is done, a report will be written, the Planner reads it, and if vulnerability detections are reported, it will call the results analyst to analyze these results and write a report in natural language to present to the user, as can be seen in Figure 1 down below.

3.3 Agent Manager

The interaction with the system can occur in two forms:

- (i) The automated pipeline that generates the Prolog reports
- (ii) The agent manager module that uses the BMAD[12] framework, to organize and find agent description files, an interactive multi-agent orchestrator that leverages a local language model (Ollama or LM Studio) as its reasoning engine. It acts as an alternative when agent-integrated IDEs are not available.

The Agent Manager main program is called `IOwrapper`(standing for Input/Output wrapper) it manages the agent's context stack, where each agent has its own identity and conversation history. The user can switch between specialist agents—such as the Planner and Results Analyst during the same session while preserving the state of each agent using the context stack.

Each LLM call follows a fixed template containing the agent identity, tool definitions, and formatting instructions. LLM responses may include tool calls delimited by special tags developed for this purpose, which are interpreted and executed sequentially by `IOwrapper`. The available tools comprise:

- `read_file` and `write_in_file`;
- `run_command` and `list_files`;
- `find_file_path` and `get_current_time`;
- `switch_agent` and `return_to_previous_agent`.

Operations that modify files or execute commands require explicit confirmation.

The execution flow consists of a two-level reactive loop:

- **Level 1:** Main user interaction loop, responsible for reading the user input and forwarding it to the current agent.
- **Level 2:** Reaction loop (up to 10 iterations), where LLM responses are processed. If tool calls are detected, they are executed, and the results are sent back to the LLM; otherwise, the response is presented as the final output.

Two tools are treated as meta-tools and immediately interrupt the reaction loop: `switch_agent`, which pushes a new agent context onto the stack, and `return_to_previous_agent`, which pops the current context and restores the previous one. This mechanism allows users to delegate subtasks to specialist agents and then return to the main agent with the generated results.

`IOwrapper` does not replace the symbolic pipeline:

`facts emissor` → Prolog → `report writer`

But rather acts as an orchestration layer on top of it inside the scope of the Agent Manager. The symbolic pipeline is invoked indirectly through the `run_command` tool by the Planer agent, enabling LLM agents to coordinate analyses, interpret results, and collaboratively generate structured reports. NASC only need the two agents described in this work to function, but the BMAD[12] framework has a vast variety of built in agents for different use cases which the user may find useful.

3.4 Structural Fact Generation

The first step of the symbolic core after it is executed is to create a list of prolog facts from the AST. A fact can be seen as a declarative statement that represents an operation on the AST, so node ids, function calls and assigments are examples of what is used to generate tuples that are the Prolog facts. The Extractor traverses the Solidity AST and emits facts, building the list. It supports two AST formats — the standard Solidity compiler output and the SmartBugs-Curated[8] variant — ensuring broad dataset compatibility.

The system emits facts of arity 4 (for leaf-level attributes), which means that each fact represents a relation among four pieces of data and/or metadata, and arity 5 (for compound statements encoding Control Flow Graph edges).

In this way, the Extractor leverages the native numeric identifiers assigned to each node by the Solidity compiler on the AST. This ensures an exact 1:1 mapping of dataflow. For instance, when a variable is assigned the result of a function call, the system records an `assignment` fact using the same AST node ID as the `external_call` fact. This eliminates false positives caused by variable shadowing or scope aliasing.

The line number of each function is resolved through a four-step pipeline: (i) a regular expression over the source code to capture `function`, `constructor`, and `modifier` declarations; (ii) candidate matching by function name against fact identifiers; (iii) content-based triangulation using the Nth-statement body signature for remaining unresolved identifiers; and (iv) a byte-offset fallback when all other methods fail. This process reduces unresolved function identifiers by over 95%.

3.5 Logic Engine: Relational Inference

The Logic Engine ingests the facts database generated and applies 24 declarative detection rules, which essentially are conditions that check for specific patterns on the facts, organized across seven of the vulnerability domains annotated in the dataset of [8]:

- **Reentrancy (4 rules)** — Reentrance patterns via CFG `succ` edges.
- **Access Control (3 rules)** — Unprotected critical operations, incorrect names and `tx.origin` authentication.
- **Arithmetic (2 rules)** — Overflow in Solidity under version 0.8 and unchecked block usage in >=0.8.
- **Unchecked Low-Level Calls (5 rules)** — Delegatecall, self-destruct, send/transfer without verification, etc.
- **Denial of Service (6 rules)** — Loop-based external calls, gas exhaustion, send failure, unbounded loops, and griefing patterns.
- **Timestamp Dependence (1 rule)** — `block.timestamp` usage outside require/assert/emit contexts.
- **Compiler Warnings (2 rules)** — Floating pragma and outdated compiler versions.

Since the facts are generated from the AST and respect the tree's scopes, the list of facts can be organized in graphs that represent scopes and relations, as well as, ancestry and flux control. So we mitigate infinite checking loops by using tabling, which is a way to not check the same graph nodes repeatedly stopping the loop. This approach speeds up the analysis by 137–195× compared with previous attempts without using tabling.

Because the structural hierarchy and dataflow are preserved through native AST node IDs, the Prolog engine performs complex traversals natively using logical resolution and backtracking, without relying on heuristic source-line ordering.

3.6 Execution Pipeline

The pipeline is defined declaratively in a YAML workflow and executed, using TEA[2], by a Planner Agent — the system control node. The flow comprises four stages: (1) fact generation; (2) concatenation of all 12 modular Prolog rule files, the runner meta-predicate `check_all/1` used to apply all rules at once, and the emitted facts into a single program; (3) logic inference via the `janus_swi` bridge, executing `check_all(R)` which collects and deduplicates all detection atoms; and (4) report generation. If vulnerabilities are found, the Planner delegates to the Results Analyst Agent that enriches the raw detections into a structured natural-language audit report. In Figure 1, we can see how the symbolic pipeline works overall.

3.7 Report Generator

The Report Generator parses Prolog detection atoms and produces Markdown reports with the derivation of facts and satisfied rules that support the detection. Each detection is parsed to a `{type, function, byte_offset, extra}` record.

The final symbolic report includes the original source code with five lines of context before and after each detection, along with the Prolog facts and satisfied rules that triggered it, ensuring complete evidence traceability.

4 Experimental Setup

The quantitative evaluation focuses on the symbolic core. The systematic evaluation of the LLM-based layers is discussed as a limitation in Discussion and Limitations Section.

To evaluate the efficacy of the symbolic core, we tested the system across three distinct smart contract datasets: SmartBugs-Curated[8], MANDO-HGT[14], and DAppSCAN[16], totaling 1,666 contracts.

SmartBugs-Curated[8] is a manually curated benchmark comprising 143 Solidity smart contracts collected from GitHub repositories, deployed contracts, and other sources, labeled according to the DASP classification. MANDO-HGT[14] is a dataset that provides real Ethereum contracts labeled across 7 types of vulnerabilities. The contracts are included with precomputed control-flow graphs (CFGs) and call graphs (CGs). The DAppSCAN[16] is a large-scale dataset sourced from professional security audits of real-world projects with hundreds of smart contracts annotated with the SWC classification. The SWC classification is a well known smart contract weakness classification registry as cited by [16], the SWC registry is specific for the ethereum platform. The combination of these datasets aims to obtain a more diverse collection of contracts, improving the representativeness. Bellow we can see a comprehensive list of all types of vulnerabilities they cover.

(1) SWC-135: Code With No Effects
(2) SWC-101: Integer Overflow and Underflow
(3) SWC-107: Reentrancy
(4) SWC-104: Unchecked Call Return Value
(5) SWC-102: Outdated Compiler Version
(6) SWC-103: Floating Pragma
(7) SWC-128: DoS With Block Gas Limit
(8) SWC-114: Transaction Order Dependence
(9) SWC-100: Function Default Visibility
(10) SWC-131: Presence of Unused Variables
(11) SWC-116: Block Values as a Proxy for Time
(12) SWC-105: Unprotected Ether Withdrawal
(13) SWC-108: State Variable Default Visibility
(14) SWC-119: Shadowing State Variables
(15) SWC-113: DoS with Failed Call
(16) SWC-129: Typographical Error
(17) SWC-120: Weak Sources of Randomness from Chain Attributes
(18) SWC-123: Requirement Violation
(19) SWC-112: Delegatecall to Untrusted Callee
(20) SWC-134: Message Call with Hardcoded Gas Amount
(21) SWC-126: Insufficient Gas Griefing
(22) SWC-124: Write to Arbitrary Storage Location
(23) SWC-111: Use of Deprecated Solidity Functions
(24) SWC-115: Authorization through tx.origin
(25) SWC-110: Assert Violation
(26) SWC-122: Lack of Proper Signature Verification
(27) SWC-125: Incorrect Inheritance Order
(28) SWC-117: Signature Malleability
(29) SWC-133: Hash Collisions With Multiple Variable Length Arguments
(30) SWC-121: Missing Protection against Signature Replay Attacks
(31) SWC-118: Incorrect Constructor Name
(32) SWC-106: Unprotected SELFDESTRUCT Instruction
(33) SWC-132: Unexpected Ether Balance
(34) SWC-109: Uninitialized Storage Pointer
(35) SWC-136: Unencrypted Private Data On-Chain
(36) SWC-130: Right-To-Left-Override Control Character (U+202E)
(37) SWC-127: Arbitrary Jump with Function Type Variable

NASC covers 15 of these 37 vulnerabilities, though we use a different nomenclature, we still mapped back to the SWC nomenclature to contextualize our recall results with the tools analyzed by [16], from their nomenclature NASC can detect: SWC-100 (Function-Level), SWC-105 (Unprotected Ether), SWC-106 (Unprotected SELFDESTRUCT), SWC-108 (State Variable Default), SWC-115 (tx.origin), SWC-118 (Incorrect Constructor),SWC-101 (Overflow/Underflow), SWC-107 (reentrancy),SWC-104 (Unchecked Call), SWC-112 (Delegatecall), SWC-134 (Hardcoded Gas), SWC-113 (DoS), SWC-126 (Gas Griefing), SWC-128 (DoS with Block Gas) and SWC-116 (Timestamp).

4.1 Challenge: Contract-Level to Line-Level Evaluation

Early evaluations used contract-level matching: if a contract was labeled with a vulnerability category and TEA[2] detected that category anywhere in the contract, it was counted as a True Positive. This approach proved misleading. A contract labeled `reentrancy` might be wrongly marked as detected even though the rule actually mapped to a wrong function that had no relation to the actual vulnerability. This inflated the metrics and gave off a false sense of what the architecture could really detect. Another issue that descended from this was the fact that a large number of contracts in the MANDO-HGT[14] dataset had many occurrences of vulnerabilities annotated per contract; all of these annotations would be reduced to a single one. Because of this, we considered that approach essentially an error and proceeded to correct it in order to obtain clearer and more objective metrics. This issue is also reported in the historical evolution section, showing the fall in recall after the fix.

To address this, we implemented fine-grained line-level matching. The ground truth includes exact source lines for each vulnerability. A detection is considered a true positive only if its line falls within a threshold of five lines of a ground-truth annotation of the same category. This required solving the function-identifier-to-line-number mapping problem, which reduced unresolved identifiers by over 95% across the SmartBugs-Curated dataset.

4.2 Evaluation Metrics

Each detection is classified by comparing its source line against the ground-truth annotation lines of the same vulnerability category within the same contract. A detection is a True Positive (TP) if its line falls within a threshold of five lines of any ground-truth line of the matching category. A detection with no matching ground truth within the threshold is a False Positive (FP). Ground-truth lines that are not covered by any detection are False Negatives (FN). Contracts without an annotation for a given category are counted

Table 2: Line-Level Detection Performance across Datasets

Dataset	Category	TP	FP	FN	Precision	Recall	Accuracy	F1
	access_control	15	205	9	6.8%	62.5%	6.6%	12.3%
	arithmetic	22	427	1	4.9%	95.7%	4.9%	9.3%
	denial_of_service	5	256	9	1.9%	35.7%	1.9%	3.6%
SmartBugs-Curated	reentrancy	30	235	2	11.3%	93.8%	11.2%	20.2%
	time_manipulation	5	55	2	8.3%	71.4%	8.1%	14.9%
	unchecked_low_level_calls	66	101	9	39.5%	88.0%	37.5%	54.5%
	Micro-avg	**143**	**1,279**	**32**	**10.1%**	**81.7%**	**9.8%**	**17.9%**
	access_control	927	7,751	407	10.7%	69.5%	10.2%	18.5%
	arithmetic	1,334	3,013	17	30.7%	98.7%	30.6%	46.8%
	denial_of_service	1,037	6,256	338	14.2%	75.4%	13.6%	23.9%
MANDO-HGT	reentrancy	1,341	4,609	3	22.5%	99.8%	22.5%	36.8%
	time_manipulation	905	118	476	88.5%	65.5%	60.4%	75.3%
	unchecked_low_level_calls	1,265	5,422	5	18.9%	99.6%	18.9%	31.8%
	Micro-avg	**6,809**	**27,169**	**1,246**	**20.0%**	**84.5%**	**19.3%**	**32.4%**
	access_control	12	344	199	3.4%	5.7%	2.2%	4.2%
	arithmetic	87	878	125	9.0%	41.0%	8.0%	14.8%
	denial_of_service	63	921	140	6.4%	31.0%	5.6%	10.6%
DAppSCAN	reentrancy	45	2,822	110	1.6%	29.0%	1.5%	3.0%
	time_manipulation	22	218	43	9.2%	33.8%	7.8%	14.4%
	unchecked_low_level_calls	27	214	142	11.2%	16.0%	7.0%	7.7%
	Micro-avg	**256**	**5,397**	**759**	**4.5%**	**25.2%**	**4.0%**	**7.7%**

as True Negatives (TN) when the system also produces no detection; if it does produce a detection, then it is a False Positive (FP).

5 Results

In this work we only evaluated the symbolic core, we did not evaluate the LLM aspect of NASC. Further explanations can be found on the Discussion and Limitations Section.

5.1 Performance Across Datasets

Table 2 shows the per-category performance on the SmartBugs-Curated dataset[8]. The system achieves high recall on arithmetic (95.7%), reentrancy (93.8%), and unchecked low-level calls (88.0%), but lower recall on denial of service (35.7%) and timestamp manipulation (71.4%), where the Solidity patterns are more diverse. The micro-averaged recall of 81.7% demonstrates that the symbolic core reliably identifies most annotated vulnerability lines in this dataset.

The MANDO-HGT[14] dataset achieves a micro-averaged recall of 84.5% and an F1 of 32.4%. It is important to note that MANDO-HGT contracts have a high density of annotated vulnerability lines. In this case, some contracts mark over 50 lines as vulnerable within a single category. This dense annotation increases the probability of line-level matches within the threshold and partially explains the higher recall and F1 score in comparison with the other datasets as seen in Table 2.

The DAppSCAN[16] dataset shows lower recall (25.2% micro-average), reflecting the complexity of real-world audited contracts, which sometimes involve business-logic vulnerabilities and multi-contract interactions that fall outside the scope of the structural patterns detectable by the current rule set. Additionally, 57 contracts (4.9%) failed during AST traversal, and 39 of the 806 ground-truth contracts lacked corresponding reports due to mapping gaps. The specific metrics can be seen in Table 2.

5.2 Contextualization with Dappscan Benchmark

DappSCAN[16] dataset uses a different nomenclature for vulnerabilities as briefed in the Experimental Setup Section. It is more specific per vulnerability type. There are 37 vulnerability types defined by them, but in the dataset used there are 35 types annotated. They all use the format SWC.

To situate our results within the existing literature, Table 3, down below, contextualize TEA against the seven tools evaluated in the original DAppSCAN[16] benchmark paper across all SWC vulnerability types present in the dataset. Table 3 reproduces the original benchmark results and adds TEA line-level metrics using the sum of all lines marked as vulnerable per SWC category which were computed using the **Sum of all Lines marked in the ground truth as vulnerable** per each vulnerability, meaning if a single vulnerability is marked as a range of 5 consecutive lines those 5 lines will be counted as 5 detections. So the differences in metrics from this table to Table 2 are due to this discrepancy in counting approach, as we can see in the rise in precision and the fall in recall, since multiple detections in a line range would be added to the sum and a single miss could mean missing the entire range of the annotated vulnerability. We chose this different approach to preserve granularity as much as possible of our evaluation. The tools tested in [16] use detections per file, according to the evaluation protocol adopted in their benchmark. We considered comparing per file but,

Table 3: DAppSCAN Benchmark and TEA-Symbolic core Results.

SWC	Mythril TP/FP/FN	Securify TP/FP/FN	Slither TP/FP/FN	Smartian TP/FP/FN	Sailfish TP/FP/FN	eTainter TP/FP/FN	Echidna TP/FP/FN	TEA TP/FP/FN
100	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	5/17/248
101	2/71/90	/ / /	/ / /	0/105/92	/ / /	/ / /	1/18/91	190/367/1199
102	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	—
103	/ / /	37/1298/53	82/4132/8	/ / /	/ / /	/ / /	/ / /	—
104	0/8/60	0/114/60	13/538/47	0/26/60	/ / /	/ / /	/ / /	16/24/225
105	/ / /	0/6/14	0/56/14	0/4/14	/ / /	/ / /	/ / /	7/10/492
106	0/20/1	0/8/1	0/15/1	0/12/1	/ / /	/ / /	/ / /	0/0/15
107	15/115/68	1/13/82	45/459/38	0/0/83	0/31/83	/ / /	/ / /	90/277/497
108	/ / /	0/124/4	/ / /	/ / /	/ / /	/ / /	/ / /	0/16/49
109	/ / /	0/322/1	0/59/1	/ / /	/ / /	/ / /	/ / /	—
110	0/106/0	/ / /	/ / /	0/350/0	/ / /	/ / /	/ / /	—
111	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	—
112	0/8/6	0/1/6	0/19/6	/ / /	/ / /	/ / /	/ / /	0/0/358
113	0/26/7	/ / /	/ / /	0/4/7	/ / /	0/0/7	/ / /	77/35/123
114	/ / /	1/37/41	/ / /	/ / /	1/56/41	/ / /	/ / /	—
115	0/10/1	/ / /	1/1/0	0/2/1	/ / /	/ / /	/ / /	0/8/8
116	0/126/3	0/28/3	0/305/3	0/6/3	/ / /	/ / /	/ / /	21/15/40
117	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	—
118	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	0/1/2
119	/ / /	0/40/0	0/331/0	/ / /	/ / /	/ / /	/ / /	—
120	/ / /	/ / /	0/24/5	/ / /	/ / /	/ / /	/ / /	—
121	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	—
122	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	—
123	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	—
124	0/1/0	0/248/0	/ / /	0/5/0	/ / /	/ / /	/ / /	—
125	/ / /	/ / /	0/72/1	/ / /	/ / /	/ / /	/ / /	—
126	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	11/25/72
128	/ / /	/ / /	1/167/48	/ / /	/ / /	0/10/49	/ / /	76/103/191
131	/ / /	0/232/44	2/222/42	/ / /	/ / /	/ / /	/ / /	—
133	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	—
134	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	/ / /	0/13/15
135	/ / /	/ / /	105/2735/30	/ / /	/ / /	/ / /	/ / /	—

TEA overall: TP = 493, FP = 946, FN = 3,307, Precision = 34.3%, Recall = 13.0%.

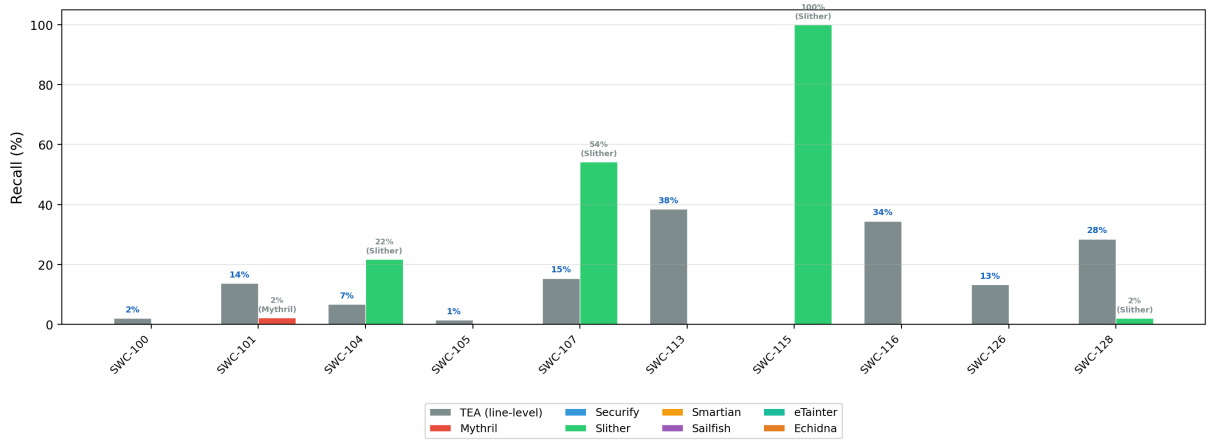

Figure 2: Line-level recall contextualization with NASC and the best-performing tool from the DAppSCAN dataset.

if we did so, we would lose localization information, since line-level metrics better capture its vulnerability localization capability than per file metrics.

SWC types that do not map to any of the six TEA vulnerability categories are marked as "—".

Of the 35 SWC types present in the DAppSCAN dataset, NASC can detect 15 that map to its six vulnerability categories. On several types where the benchmark tools achieve zero or near-zero recall, such as SWC-113 (DoS with Failed Call), SWC-116 (Block Values as Proxy for Time), and SWC-128 (DoS With Block Gas Limit), TEA achieves line-level recall of 28.5%–38.5% . The structural nature of TEA's Prolog rules (in-scope data-flow taint, loop safety, barrier analysis) shows particularly strong performance for denial-of-service(SWC-128) and timestamp-related patterns(SWC-116), which are underrepresented in other static analysis tools, as can be seen in the Table 3. In Figure 2, we can observe a set of 10

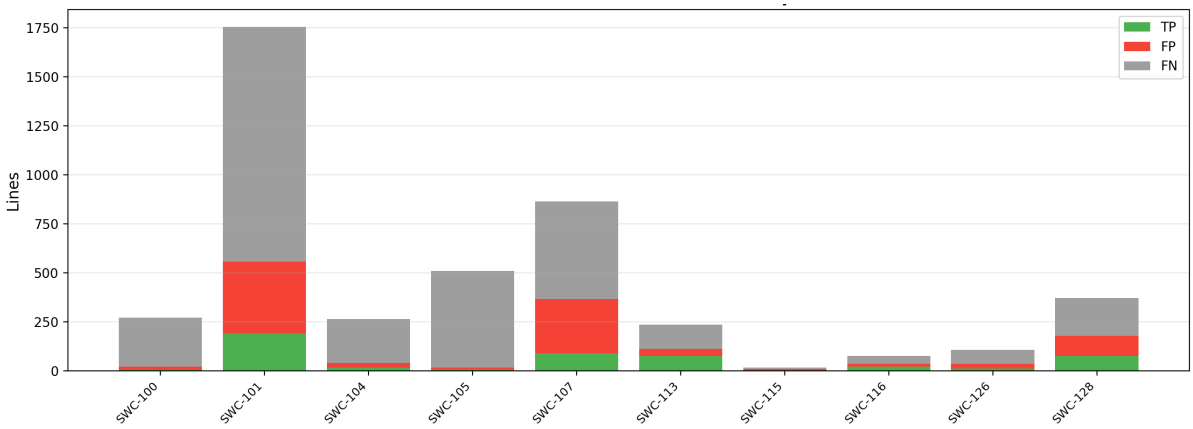

Figure 3: Detailed performance metrics breakdown for TEA-Symbolic core across evaluation categories.

Table 4: Detection Volume Before and After Duplicate Collapse.

Dataset	Reports	Before Collapse	After Collapse	Reduction
SmartBugs-Curated	142	2,108	1,667	20.9%
MANDO-HGT	350	39,722	35,169	11.5%
DAppSCAN	676	26,544	13,492	49.2%

Table 5: Evolution across NASC versions.

Version	TP	FP	FN	Recall
v1 (Procedural heuristic)	104	278	23	81.9%
v2 (CFG integration)	112	280	15	88.2%
v2.5 (Native-ID + transitivity)	124	310	3	97.6%
v2.8 (Tabling + 4-step line resolution + collapse)	143	1,279	32	81.7%

vulnerabilities covered by NASC in a Recall contextualization with the best tool in each vulnerability category from the DappSCAN [16] article per vulnerability type, where we can see that, although the evaluation protocols differ, we can observe our recall in context with the best tool per vulnerability from the DappSCAN[16], **keeping in mind that our evaluation is line-level and theirs is file level**, so **these values should be interpreted as contextual rather than directly comparable**. In Figure 3, we can see the breakdown of TP, FP, and FN per vulnerability and number of detections.

5.3 Total Detection Volume

Table 4 quantifies the total number of detection rows generated across all datasets, before and after the post-processing collapse.

Approximately 17.8% of the raw detection rows correspond to types that are not mappable to any of the six evaluated vulnerability categories (e.g., `UNHANDLED INTERFACE CALL`, `OUTDATED COMPILER`). These are excluded from the metrics but are present in the intermediate reports.

5.4 Historical Evolution

Table 5 captures the progressive improvements in baseline resolution metrics across developmental phases. The project's architectural iterations are outlined chronologically regarding the SmartBugs-Curated dataset.

Table 5 traces the evolution of the architecture. The transition from v2.5 to v2.8 represents a methodological shift from contract-level to line-level evaluation. The apparent reduction in recall (97.6% to 81.7%) is not a regression but the correction of the misleading contract-level metric: v2.5 counted detections that matched the contract's label but not the specific annotated line. The v2.8 line-level metric is stricter and more truthful. Key technical milestones along this trajectory include: (i) eliminating 188 timeouts through `:- table` directives (137–195× speedup); (ii) reducing unresolved function identifiers from 43 to 5 through the four-step line resolution pipeline; and (iii) collapsing 441 duplicate detection rows in SmartBugs-Curated alone.

6 Discussion and Limitations

Symbolic core only evaluation. In this work we only evaluated the symbolic core, as it's results can be assessed objectively using

the ground-truth annotations. We took this decision to maintain our evaluation as objective as possible. To evaluate the LLM components we would need to prepare a separate experimental setup. To fairly evaluate the results generated by the LLM we would need to conduct a quality evaluation with other developers or security analysts of the area, to account for metrics such as clarity, cohesion, correctness and usefulness which is beyond the scope of this work.

Precision and the Structural Nature of Static Analysis. The precision observed across the evaluated datasets is relatively low (10.1% micro-average on SmartBugs-Curated and 4.5% on DappSCAN), while recall remains higher. This behavior is consistent with the design trade-offs commonly observed in syntax- and rule-based static analysis tools, which prioritize detecting as many potentially vulnerable patterns as possible, even at the cost of producing additional false positives as show in the DappSCAN[16] paper, in table 3 and Figure 2. Since the analysis is performed over AST-derived facts and control-flow relationships without full semantic reasoning or runtime information, structurally similar code fragments may satisfy vulnerability rules despite not being exploitable in practice. For example:

- Arithmetic: Unchecked incremental variables in loops may be flagged even when the program semantics do not lead to exploitable behavior.
- Reentrancy: Contracts with external calls followed by state updates are flagged, regardless of whether exploitation is actually possible under execution constraints.
- Denial of service: Patterns can be triggered in potentially unlimited loops or externally influenced control flow, even when actual conditions prevent reaching a vulnerable state.

The impacts of this can vary per use case. The metrics were collected using hundreds of contract files. NASC is not well suited for large-scale automated screening where manual inspection of every finding is impractical. It would be useful as an assistant tool though, on a selected smaller set of contracts, where a human reviewer can inspect and validate the reported findings.

To avoid these false positives one needs to either run a fuzzy detector or mutation test, as they introduce execution data in the analysis. This highlights the importance of semantic and fuzzy approaches, which can work precisely in these weaknesses.

MANDO-HGT Annotation Density. The MANDO-HGT dataset achieves the highest F1 (32.4%) and recall (84.5%), but this must be contextualized by its annotation density. Contracts in this dataset frequently mark dozens of lines per category as vulnerable. With a threshold of five lines, the probability of a match increases proportionally to the annotated surface area. While the detections are structurally correct, the high recall may partially reflect favorable annotation coverage rather than superior detection capability.

DAppSCAN Real-World Complexity. The DAppSCAN dataset consists of contracts from professional security audits. These contracts exhibit complex business logic, multi-contract inheritance, and vulnerabilities that go beyond syntactic patterns (e.g., economic incentive mismatches, governance manipulation). The current symbolic rule set operates entirely on structural AST features (reads, writes, calls, control flow), making it ill-suited for semantic vulnerabilities. The 25.2% recall reflects this limitation.

Duplicate Inflation. A significant fraction of the raw detection volume consists of multiple atoms generated by the same rule for the same line. In DAppSCAN[16], 74.7% of the 26,544 raw mappable detection rows are duplicates collapsed by the line-level metric's set-based duplication. The post-processing collapse makes the intermediate reports more readable but does not affect the metrics.

Degraded AST Resilience. During batch evaluation, 57 out of 1,173 [16] database contracts (4.9%) yielded ASTs with empty `subNodes` due to legacy parsing tools. The extractor handles these by emitting explicit statements, isolating parsing failures from logical inference. This is a reduction from an earlier estimate of 16.6% due to improvements in AST regeneration.

7 Conclusion and Future Work

In this paper, we introduced NASC, a decoupled Neuro-symbolic architecture that advances the reliability, precision, and explainability of smart contract auditing. NASC bridges the comprehension barrier by splitting the execution workload into two distinct layers: a deterministic symbolic core that guarantees precise vulnerability matching, and a neural multi-agent layer that translates these logical proofs into comprehensive, human-readable security reports.

Validation across three distinct datasets demonstrates that this neuro-symbolic synergy can act as an assistant in the security comprehension barrier for structurally detectable vulnerability categories. However, our findings also illuminate critical challenges that define the path forward for automated blockchain security analysis. Future work will focus on scaling our approach's precision, expanding automated dataset annotation coverage, and deepening the neural agents' capacity for complex semantic and behavioral reasoning in multi-contract financial ecosystems.

Artifact Availability

The complete source code of our architecture, including the symbolic module with the Prolog rules and the agents definition module is publicly available on GitHub repository https://github.com/cleidimar-passos/Neurosimbolic-agent-for-smart-contracts.

Acknowledgements

This study was supported by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Brazil, under Finance Code 001. The Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG) and the National Council for Scientific and Technological Development (CNPq) also supported this work.

References

[1] Tegan Brennan, Ruoyu Zhang, Arnab Banerjee, and Manuel Egele. 2022. Sailfish: Vetting smart contract state-inconsistency bugs via graph-based-and-property-directed symbolic execution. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1211–1226.

[2] Fabio Ceolin. 2026. *The Edge Agent: A Neurosymbolic Framework for Autonomous Edge AI Agents.* https://github.com/fabceolin/the_edge_agent

[3] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Benjamin Livshits. 2024. Smart contract and defi security tools: Do they meet the needs of practitioners?. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[4] Jaeseung Choi, Joonun Kim, Byungho Kim, Jiho Shin, and Sang Kil Cha. 2021. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analysis. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 548–560.

[5] Young-Woo Choi, Tae-Eun Kim, Jin-Woo Hong, Chang-Seo Song, and Hyoung-Shik Kim. 2023. eTainter: Detecting Gas-Guzzling Vulnerabilities in Smart Contracts. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[6] ConsenSys Diligence. 2026. Mythril: Security analysis tool for Ethereum smart contracts. https://github.com/ConsenSys/mythril

[7] 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).

[8] Monika di Angelo, Thomas Durieux, João F. Ferreira, and Gernot Salzer. 2023. SmartBugs 2.0: An Execution Framework for Weakness Detection in Ethereum Smart Contracts. arXiv:2306.05057 [cs.CR] https://arxiv.org/abs/2306.05057

[9] 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.

[10] Gustavo Grieco, Alex Groce, Josselin Feist, and Trail of Bits. 2020. Echidna: A Fast Smart Contract Fuzzer. https://github.com/crytic/echidna

[11] Daojing He, Zhi Deng, Yuxing Zhang, Sammy Chan, Yao Cheng, and Nadra Guizani. 2020. Smart contract vulnerability analysis and security audit. *IEEE Network* 34, 5 (2020), 276–282.

[12] Brian Madison and BMad Code Org. 2026. *BMAD-METHOD: Breakthrough Method for Agile Ai Driven Development.* Acessado em: 2026-06-30.

[13] Uzma Nawaz, Mufti Anees-ur Rahaman, and Zubair Saeed. 2025. A review of neuro-symbolic AI integrating reasoning and learning for advanced cognitive systems. *Intelligent Systems with Applications* 26 (2025), 200541.

[14] Hoang H. Nguyen, Nhat-Minh Nguyen, Chunyao Xie, Zahra Ahmadi, Daniel Kudendo, Thanh-Nam Doan, and Lingxiao Jiang. 2023. MANDO-HGT: Heterogeneous Graph Transformers for Smart Contract Vulnerability Detection. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 334–346. doi:10.1109/MSR59073.2023.00052

[15] Hadis Rezaei, Mojtaba Eshghie, Karl Andersson, and Francesco Palmieri. 2025. SoK: Root Cause of $1 Billion Loss in Smart Contract Real-World Attacks via a Systematic Literature Review of Vulnerabilities. *arXiv preprint arXiv:2507.20175* (2025). arXiv:2507.20175 [cs.CR] https://arxiv.org/abs/2507.20175

[16] Zibin Zheng, Jianzhong Su, Jiachi Chen, David Lo, Zhijie Zhong, and Mingxi Ye. 2024. DAppSCAN: Building Large-Scale Datasets for Smart Contract Weaknesses in DApp Projects. *IEEE Transactions on Software Engineering* 50, 6 (2024), 1360–1373. doi:10.1109/tse.2024.3383422