

Do Smart Contract Auditing Results Transfer Across Datasets? A Two-Benchmark Empirical Study of Static and LLM-Based Security Tools

Shawal Khalid ✉ 

Department of Computer Science, Virginia Tech, Blacksburg, VA, USA

Joaquin Tuckett ✉

Department of Computer Engineering, Virginia Tech, Blacksburg, VA, USA

Chris Brown ✉ 

Department of Computer Science, Virginia Tech, Blacksburg, VA, USA

Abstract

Background. Smart contract vulnerabilities pose significant security risks in blockchain systems. Automated static auditing tools are commonly used in practice, yet their effectiveness varies across vulnerability categories and at scale. Recently, large language model (LLM)-based tools have been proposed, but their performance is rarely evaluated against real-world contracts and alongside established analyzers.

Aim. This paper presents a systematic benchmarking study of smart contract auditing tools under a single, explicitly defined task: identifying smart contract vulnerabilities in Solidity contracts.

Method. We evaluate static and LLM-based tools ($n = 7$) on two benchmarks: SmartBugs Curated ($n = 143$) and a filtered FORGE subset of real-world audit-derived contracts ($n = 173$). We use a tool-agnostic evaluation pipeline to normalize heterogeneous tool outputs, reporting accuracy, top- K detection, execution robustness, and cross-dataset performance.

Results. Our findings expose systematic trade-offs across auditing approaches. Static tools optimized for broad vulnerability coverage tend to over-report categories, potentially increasing triage burden, while exhibiting stability limitations on real-world contracts. LLM-driven auditors demonstrate strong coverage across many vulnerability categories, but similarly suffer from over-prediction.

Conclusions. These results show that current smart contract auditing tools differ less in their ability to identify potential vulnerabilities than in their ability to do so precisely and reliably, motivating future auditing workflows for practical smart contract vulnerability detection.

2012 ACM Subject Classification Software and its engineering → Software verification and validation; Security and privacy → Software and application security

Keywords and phrases Smart contract vulnerability detection, static analysis, symbolic execution, benchmarking, large language models

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.45

Category Technical Track Paper

Supplementary Material *Software:* <https://github.com/shawalkhalid667/smartcontracts-auditor-benchmark>, archived at `swb:1:dir:760a95725ee07df7e02d285a70655c4516e7ae97`

Funding *Shawal Khalid:* supported by a BitShares Graduate Fellowship.

Joaquin Tuckett: supported by the Virginia Tech Mentorship and Academic Opportunities Program (MAOP) for undergraduate research.

1 Introduction

Smart contracts are a foundational component of modern blockchain ecosystems, underpinning decentralized finance (DeFi), governance protocols, and supply-chain applications [5]. Unlike traditional software, smart contracts are immutable after deployment and operate in adversarial, economically incentivized environments, where vulnerabilities can be actively



© Shawal Khalid, Joaquin Tuckett, and Chris Brown;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;
Article No. 45; pp. 45:1–45:21



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

exploited for financial gain [50]. As a result, security defects – such as reentrancy, arithmetic errors, and access-control flaws – can lead to irreversible losses and systemic failures [32]. For instance, the Decentralized Autonomous Organization (DAO) exploit in 2016 [33] led to the loss of over \$50 million (3.6 M ETH). Subsequent attacks on DeFi protocols further demonstrate smart contract security remains a critical and unresolved challenge [2].

To mitigate these risks, a wide range of automated smart contract analysis tools have been developed and adopted by developers and auditors [9, 15, 41]. These tools rely on different static analysis and symbolic execution techniques – such as rule-based detection, control-flow and data-flow analysis, and constraint solving – to identify vulnerability patterns. Such analyzers are routinely integrated into smart contract development workflows, continuous integration (CI) pipelines, and professional audits [25, 15]. Despite their widespread adoption, static analyzers exhibit fundamental and well-documented limitations. Rule-based and pattern-driven tools tend to over-approximate vulnerability conditions, leading to high false-positive rates and increased manual triage effort [12, 17]. Symbolic execution offers greater expressiveness, but suffers from path explosion, scalability challenges, and brittle behavior on complex or real-world contracts [35]. Large-scale empirical studies show that no static analyzer provides strong coverage across vulnerability classes while maintaining both precision and robustness [13].

Recent research explores the use of large language models (LLMs) for smart contract auditing [22]. Rather than relying on predefined rules or symbolic exploration, LLM-based approaches generate vulnerability predictions directly from contract code based on a corpus of training data, sometimes augmented with structured prompting or post-processing. For example, *Smart-LLaMA*, uses “smart contract-specific continual pre-training” and “explanation-guided fine-tuning” to detect and explain vulnerabilities [49]. While early results suggest such models can detect a broad range of vulnerability categories with high recall [10, 47], existing evaluations are fragmented: they often use custom datasets, inconsistent vulnerability taxonomies, or lack direct comparison against established tools. Thus, **it remains unclear how model-driven auditors perform relative to traditional tools when evaluated on the same contracts, under the same labeling scheme, and with the same metrics.**

Our paper addresses this gap by presenting a systematic study comparing static and LLM-assisted smart contract vulnerability detection tools. We evaluate seven widely used and emerging auditing tools on two benchmarks: SmartBugs Curated ($n = 143$) [11] and a curated FORGE subset of real-world audit-derived contracts ($n = 173$) [6]. This two-dataset design allows us to retain the reproducibility and comparability of SmartBugs while directly evaluating LLM and non-LLM-based approaches – aligning with guidelines for empirical software engineering studies with LLMs [3]. To account for differences in tool capabilities, we normalize outputs into a common vulnerability taxonomy, enabling direct category-level comparison across heterogeneous reporting formats, and report coverage-aware metrics restricted to supported vulnerability categories. Beyond detection accuracy, we also examine operational robustness by measuring crashes, empty outputs, and missing or invalid results when executed across the datasets under consistent execution conditions.

To guide our study, we investigate the following research questions (RQs):

- **RQ1:** How accurately do existing smart contract auditing tools detect SmartBugs vulnerability categories under a unified evaluation protocol?
- **RQ2:** How do auditing tools differ in their effectiveness for ranked vulnerability triage, as measured by top- K detection performance?
- **RQ3:** How robust are smart contract auditing tools when executed at scale across a diverse set of real-world contracts?

■ **RQ4:** To what extent does tool performance transfer to newer real-world smart contracts?

Our findings reveal systematic trade-offs across auditing approaches. Tools that emphasize broad detection coverage tend to over-report vulnerability categories, while more expressive analyses improve detection for certain vulnerability classes at the cost of scalability and stability. LLM-driven auditors tend to exhibit similar over-prediction behavior, indicating false positives and robustness remain bottlenecks in automated smart contract auditing [7]. More importantly, our cross-dataset results show that benchmark choice can substantially change tool conclusions – suggesting that strong performance on canonical benchmarks does not necessarily transfer to newer real-world contracts. Our contributions include:

- We conduct a two-dataset empirical benchmark of smart contract auditing tools across SmartBugs Curated and FORGE.
- We propose and apply a unified evaluation protocol that normalizes tool outputs into vulnerability categories.
- We compare seven static and LLM-based tools on detection accuracy, top- K triage effectiveness, coverage-aware performance, and operational robustness.

2 Background and Related Work

Smart Contracts and Vulnerability Classes

Smart contracts are blockchain programs that automatically execute transactions and enforce protocol rules. Because deployed contracts are difficult to modify and operate in adversarial, economically incentivized environments, vulnerabilities can lead to irreversible financial loss, denial of service, or systemic protocol failures [8]. Prior work identifies recurring vulnerability classes, including reentrancy, unchecked low-level calls, access-control flaws, arithmetic errors, timestamp dependence, and insecure randomness [45]. The SmartBugs vulnerability taxonomy has become a widely used benchmark-oriented classification for organizing such vulnerabilities into comparable categories [11]. Prior work also shows that blockchain development is shaped by distinctive software-engineering practices and by the broader cryptocurrency ecosystem [21, 24, 23].

Automated Smart Contract Analysis Paradigms

To mitigate smart contract security risks, a variety of automated analysis paradigms have been developed. These approaches differ significantly in expressiveness, scalability, robustness, and the form of their reported outputs. Rule-based and linting tools are fast and easy to integrate into development workflows but often over-approximate vulnerability conditions, leading to false positives [51]. Static analyzers based on intermediate representations or control-flow graphs provide deeper structural reasoning, while symbolic execution tools explore possible execution paths but may suffer from path explosion, timeouts, or incomplete exploration on complex contracts [18]. More recently, LLM-based auditors have been proposed to predict vulnerability categories directly from contract code, but their outputs may be weakly ranked, over-predictive, or insufficiently grounded in program semantics [19].

Challenges in Evaluating Smart Contract Auditing Tools

Fair evaluation of smart contract auditing tools is difficult for three reasons. First, tools produce heterogeneous outputs, ranging from structured detector identifiers to unstructured textual reports, requiring normalization into a common representation [51]. Second, tools support different vulnerability categories, so missed detections must be distinguished from

unsupported categories [12]. Third, robustness and ranking are often under-reported: tools may crash, time out, produce empty outputs, or emit many unranked findings, even when they detect relevant vulnerabilities [37]. These issues motivate evaluation protocols that report accuracy, supported-category performance, ranked triage effectiveness, and execution robustness together.

Related Work

To enable systematic evaluation of automated analyzers, several benchmark frameworks and datasets have been proposed. SmartBugs introduced a curated dataset of vulnerable Solidity contracts and a unified framework for comparing analyzers under shared ground-truth labels [11]. Building on this infrastructure, Durieux et al. evaluated automated smart contract analysis tools at scale and showed substantial variation in detection accuracy, false positives, and robustness [13]. These studies provide important baselines, but they primarily focus on deterministic tools and canonical benchmarks.

Recent learning-based and LLM-based auditors expand this design space by predicting vulnerabilities directly from contract code, sometimes using domain-specific pretraining, fine-tuning, or program-analysis support [39, 19]. Tools such as Smart-LLaMA, FTSmartAudit, and GPTScan illustrate this shift toward model-driven or hybrid LLM-based auditing [49, 48, 40]. However, existing evaluations often rely on custom datasets, limited baselines, or heterogeneous vulnerability definitions, making it difficult to assess whether strong benchmark performance transfers to newer real-world contracts; for instance, high scores on curated benchmarks can be achieved using only word counts, without detecting any actual vulnerability [36]. In contrast to prior work, our study evaluates deterministic and LLM-based smart contract auditing tools under a unified category-level protocol across two benchmarks: SmartBugs Curated and a filtered FORGE subset.

3 Methodology

Study Design

We conducted a controlled benchmarking study of automated smart contract auditing tools, focusing on the task of *contract-level vulnerability category detection*. Given a Solidity smart contract, a tool reports one or more vulnerability classes present in the contract. This formulation reflects a common first-pass auditing workflow, where automated tools are used to triage contracts and identify likely vulnerability types prior to deeper manual analysis. We evaluate tools across two benchmark settings: a canonical benchmark used widely in prior work and a real-world audit-derived benchmark. This design allows us to assess not only tool effectiveness within a shared taxonomy, but also whether conclusions drawn from canonical benchmarks generalize to newer contracts.

Dataset

We evaluate tools on two datasets: SmartBugs Curated and a filtered subset of FORGE. **SmartBugs Curated** consists of 143 Solidity smart contracts annotated with known vulnerability categories under a fixed taxonomy. The dataset is widely used in prior empirical studies and provides manually curated ground truth labels. For evaluation, vulnerability labels are extracted from the dataset’s annotations and are used without modification for all tools. To evaluate generalization beyond SmartBugs, we also construct a FORGE subset of 173 real-world audit-derived Solidity contracts. **FORGE** provides vulnerability findings

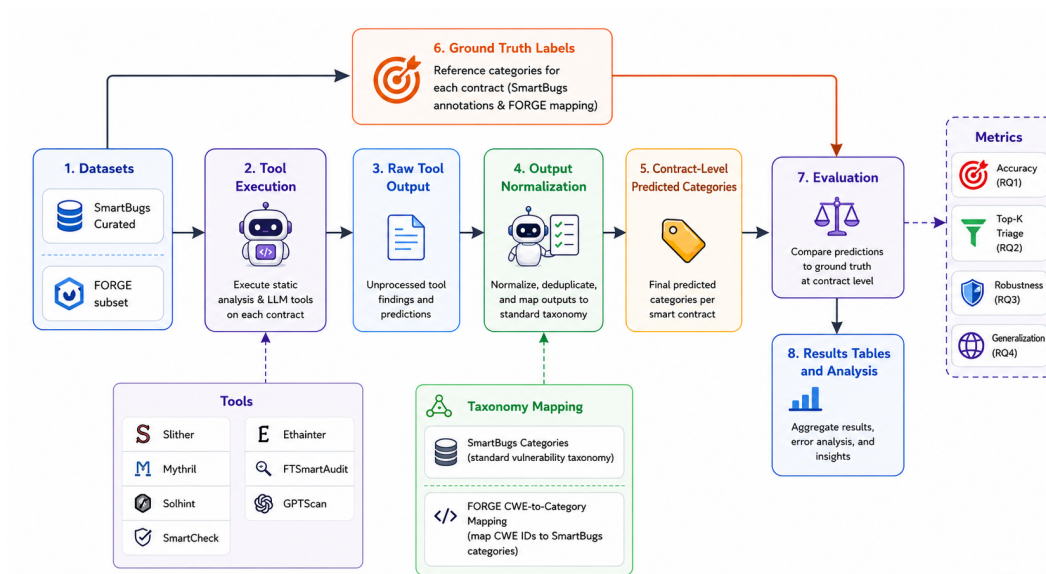
extracted from audit reports and mapped to CWE-style vulnerability descriptions. We filtered FORGE to align with our contract-level category detection task and to enable comparison with SmartBugs-style vulnerability categories. Specifically, we retained only Medium, High, and Critical severity findings; valid Solidity files; findings whose CWE labels could be mapped to our selected SmartBugs-compatible categories; and affected Solidity files with at most 500 lines of code. We excluded Low and Informational findings, non-security CWEs, and files that did not contain analyzable Solidity source code. The final FORGE subset covers seven vulnerability categories: reentrancy, unchecked low-level calls, access control, arithmetic, bad randomness, time manipulation, and denial of service. This subset provides a complementary real-world benchmark for assessing whether tool behavior observed on SmartBugs transfers to newer contracts. We evaluate detection at the category level rather than line-level localization, as the primary goal of this study is to compare tools' ability to correctly identify vulnerability classes rather than pinpoint exact source locations. For both datasets, each contract is represented by the set of ground-truth vulnerability categories associated with that contract. Tool predictions are scored against these category-level labels.

The two datasets also differ substantially in Solidity compiler versions. SmartBugs Curated primarily consists of legacy Solidity contracts, with prior work reporting that over 90% of its contracts use Solidity 0.4+. In contrast, FORGE is more representative of newer smart contract development: the full FORGE dataset reports Solidity 0.8+ as the most prevalent compiler family, accounting for 59.0% of projects. Thus, FORGE provides a useful complementary benchmark for assessing whether tool behavior on legacy canonical contracts transfers to newer real-world contracts. Most contracts in our filtered FORGE subset use Solidity 0.8.x, whereas SmartBugs Curated is dominated by legacy Solidity 0.4.x contracts.

Tool Selection

We selected a sample of traditional and LLM-based smart contract vulnerability detection tools. A comparison of all the tools observed is provided in Table 1.

- **Relevance:** Tools had to support smart contract vulnerability detection for Ethereum-based smart contracts written in Solidity. We conducted a lightweight literature review of recent academic work on smart contract security. We searched Google Scholar and major software engineering and security venues including ICSE, ISSTA, ASE, CCS, USENIX Security, using terms such as “*smart contract vulnerability detection*,” “*Ethereum security analysis*,” “*Solidity static analysis*,” “*symbolic execution smart contracts*,” and “*LLM-based smart contract auditing*.” We reviewed papers that introduced or evaluated tools for smart contract vulnerability detection, resulting in an initial set of 19 tools spanning various vulnerability detection approaches, including: static analysis ($n = 7$), symbolic execution ($n = 2$), logic-based ($n = 1$), dynamic analysis ($n = 1$), hybrid static/dynamic analysis ($n = 1$), neural networks ($n = 2$), and LLM-based ($n = 5$) approaches. The observed tools are summarized in Table 1. We further filtered selected tools by the following criteria.
- **Reproducibility:** Tools had to be reproducible by the research team. The researchers manually attempted to configure and execute the tool using publicly available artifacts, documentation, and repositories. Reproducibility was assessed based on the ability to install dependencies, resolve compiler requirements, execute the tool on representative contracts, and obtain interpretable outputs with limited manual intervention.
- **Compatibility:** Tools had to be compatible with Solidity source code analysis and our target benchmark settings. Compatibility was assessed with respect to source-code execution, compiler requirements, output interpretability, and integration into our automated



■ **Figure 1** Benchmarking pipeline for smart contract auditing tools.

pipeline. Tools that require modern compiler pipelines, runtime test harnesses, or analyze solidity versions unsupported by the dataset were excluded. For instance, MANTICORE and CONFUZZIUS require interactive fuzzing environments, failing on a substantial portion of the datasets. These exclusions are based on dataset compatibility rather than tool quality, and follow established practice in prior SmartBugs-based evaluations [13].

Static and Symbolic Analysis Tools. We include five static, symbolic, and logic-based analyzers used in prior smart contract security studies and satisfy our selection criteria. Specifically, we benchmark SOLHINT, SMARTCHECK, and SLITHER as rule-based and intermediate-representation-based static analyzers that emphasize scalability, deterministic execution, and fast triage. MYTHRIL is included to represent symbolic execution, enabling deeper semantic reasoning over EVM bytecode at the cost of execution stability. ETHAINTER is included as a logic-based semantic analyzer that applies declarative Datalog rules over Ethereum Virtual Machine (EVM) execution facts, representing a distinct analysis paradigm from both rule-based static analysis and symbolic execution.

LLM-based Tools. We include two large language model (LLM)-based auditing tools that represent complementary design choices and satisfy all screening criteria. FTSMARTAUDIT, evaluated using a single fixed foundation model (GPT-4o), represents a free-form LLM auditor that infers vulnerability categories directly from contract text without explicit structure. GPTSCAN, based on GPT-3.5-turbo, represents an LLM-assisted auditing approach that combines LLM-based code understanding with additional analysis to identify smart contract vulnerabilities.

■ **Table 1** Smart Contract Vulnerability Detection Tool Comparison.

Tool Name	Type	Input	Output	Reproducible?	Compatible?
Conkas [44]	Static	.sol, bytecode	Text	✗	✓
ConFuzzius [42]	Static, Dynamic	.sol, JSON	Text	✓	✗
Maian [31]	Static	.sol, bytecode	Text	✓	✗
Mythril [9]	Symbolic Execution	.sol, bytecode	Text, JSON	✓	✓
Ethainter [4]	Logic-based	.sol, bytecode	Text	✓	✓
Oyente [26]	Symbolic Execution	.sol, bytecode	Text	✗	✓
Securify 2.0 [43]	Static	.sol	Text	✓	✗ (~0.5.8)
Slither [15]	Static	.sol	Text, JSON	✓	✓
SmartCheck [41]	Static	.sol	Text	✓	✓
SmartEmbed [16]	Static	Text	Text	✗	✓
Solhint	Static	.sol	Text	✓	✓
Manticore [28]	Dynamic	.sol, bytecode	Workspace	✗	✗ (~0.5.12)
ContractGNN [46]	AI (GNN)	CSG	Vector	✗	✓
VulnSense [14]	AI (GNN)	.sol, bytecode	Vector	✗	✗ (~0.5.12)
FTSmartAudit [48]	LLM	Text	Text	✓	✓
GPTScan [40]	LLM	Text	JSON	✓	✓
iAudit [27]	LLM	Text	Text	✓	✗ (~0.8.0)
LLMSmartSec [29]	LLM	Text, .sol	Text	✓	✗
SmartLLM [20]	LLM	Text	Text	✗	✗ (~0.8.0)

Tools that were reproducible and compatible (highlighted) were included in our evaluation. Excluded tools are retained to demonstrate our tool selection process and clarify exclusion decisions.

3.1 Evaluation Pipeline

3.1.1 Tool Execution

We evaluate a sample of smart contract auditing tools spanning static, logic-based symbolic execution, and LLM-based analysis approaches. Each tool is executed independently on every contract using default or recommended configurations. No manual tuning, contract-specific configuration, or post hoc adjustment is applied, in order to reflect realistic out-of-the-box usage and ensure comparability across tools. All experiments are executed on both SmartBugs Curated and the filtered FORGE subset with automated scripts for tool execution, output collection, normalization, and scoring. Contracts are analyzed individually, and each tool is invoked in isolation to avoid cross-contamination of intermediate artifacts or cached results. When required, Solidity compiler versions are selected automatically using `solc-select` to match the contract’s declared pragma, subject to tool compatibility constraints.

For static, symbolic, and logic-based tools, inputs consist of Solidity source code and, where required, compiled EVM bytecode. Static analyzers (e.g., Slither, SmartCheck, Solhint) operate directly on Solidity source code or intermediate representations, while symbolic execution tools (e.g., Mythril) analyze contract behavior through path exploration over the EVM semantics. Logic-based tools (e.g., Ethainter) extract execution facts from compiled bytecode and apply declarative rules to infer vulnerability patterns.

LLM-based tools are evaluated as black-box analyzers using their publicly available implementations. Each tool is executed using its default inference configuration as provided by the authors, without modification of prompts, decoding parameters, or internal reasoning strategies. We do not perform prompt engineering, temperature tuning, or multi-turn interaction; instead, each contract is analyzed exactly once per tool via automated scripts, and only the final reported outputs are collected for evaluation. This setup reflects usage of LLM-based auditing tools as off-the-shelf systems and ensures comparability with deterministic analyzers evaluated under default configurations.

3.1.2 Output Normalization

Auditing tools produce heterogeneous outputs, including unstructured textual warnings, semi-structured reports, and structured JSON artifacts. To compare across tools, we normalize all outputs into a unified, tool-agnostic representation at the contract level. For each contract, raw tool outputs are parsed into an intermediate representation that captures individual findings, including any available detector identifiers, warning descriptions, or reported vulnerability names. These findings are then mapped to SmartBugs-compatible vulnerability taxonomy using a tool-specific mapping layer. For tools that provide explicit vulnerability labels or detector identifiers (e.g., Slither, Mythril, SmartCheck), mapping is performed using documented detector-to-category correspondences. For tools that emit free-form or semi-structured text (e.g., LLM-based tools), vulnerability categories are inferred using deterministic keyword and phrase matching based on canonical SmartBugs category names and commonly used synonyms. For FORGE, CWE-style vulnerability labels are mapped to the selected SmartBugs-compatible categories described in the dataset construction step, enabling category-level comparison across both benchmarks. When a tool reports multiple findings for a contract, individual alerts are first mapped to vulnerability categories. For micro-averaged accuracy computation, duplicate findings within the same contract-category pair are collapsed, yielding a binary indication of whether each category is detected for that contract. For ranked top- K evaluation, however, we retain category-level frequency counts so that repeatedly reported categories can be treated as stronger tool signals when explicit severity or confidence scores are unavailable. Line numbers are not used for scoring because our evaluation focuses on category-level detection rather than localization accuracy. Severity labels, confidence scores, native output order, and duplicate counts are retained only when needed to construct rankings; they do not change whether a category-level prediction is counted as correct or incorrect.

If a tool executes successfully but reports no findings, the result is recorded as an *empty output*. If a tool crashes, times out, or produces output that cannot be parsed by the normalization pipeline, the contract is marked as an *execution failure*. Empty outputs and execution failures are excluded from accuracy metrics, but are reported as part of our robustness analysis (RQ3). All normalized results are stored in a common schema that records detected categories, execution status, and tool metadata, enabling consistent computation of micro-averaged accuracy metrics, ranked top- K performance, and robustness statistics across all evaluated tools.

3.1.3 Scoring Procedure

For each tool and contract, the normalization layer produces (i) a set of detected categories and (ii) an execution status. For accuracy scoring, a true positive (TP) occurs when a detected category matches a ground-truth category for the same contract. A false positive (FP) occurs when a tool reports a category that is not present in the ground truth, and a false negative (FN) occurs when a ground-truth category is not reported by the tool. Contracts marked as execution failures are excluded from RQ1/RQ2 accuracy computations, but counted in RQ3 to measure robustness. Contracts with empty outputs are included in RQ1 as empty prediction sets and counted separately in RQ3 as empty outputs. For ranked evaluation (RQ2), category rankings are derived from tool-provided priority signals when available (e.g., severity levels or confidence scores). When tools do not provide an explicit ranking, categories are ranked by the frequency of mapped findings for that category within the contract, with more frequently reported categories treated as stronger tool signals. Ties

are resolved using the tool’s native reporting order. If fewer than K categories are emitted, we evaluate using the available categories without padding. The evaluation pipeline scripts for tool execution, output normalization, and scoring, are made available in our supplemental materials to promote the replication of this work (see Section 8).

3.2 Evaluation Metrics

3.2.1 RQ1: Accuracy

We evaluate tools on the task of contract-level vulnerability category detection. For this task, we compute **micro-averaged precision**, **recall**, and **F1 scores** across all vulnerability instances in the dataset. Micro-averaging treats each detected vulnerability instance equally and reflects realistic auditing workloads, where vulnerability categories are highly imbalanced and frequent classes dominate overall risk exposure. This avoids over-emphasizing rare categories that appear only a small number of times in the benchmark. In addition, we report the number of **true positives (TP)**, **false positives (FP)**, and **false negative (FN)** counts to outline vulnerability detection behavior and error patterns. TPs correspond to instances where detected vulnerabilities match ground-truth labels, FPs correspond to detected vulnerabilities that do not match ground-truth labels, and FNs capture ground-truth vulnerabilities that were not detected. In addition to aggregate micro-averaged metrics, we report category-level recall to examine how tool behavior varies across vulnerability categories.

Because auditing tools are designed with different detection scopes, not all tools support the same set of vulnerability categories. To account for this, we report coverage-aware metrics that restrict evaluation to categories explicitly supported by each tool. This allows us to separate detection quality from coverage limitations and ensures that tools are not penalized for vulnerabilities they are not designed to detect.

3.2.2 RQ2: Effectiveness

Automated auditing tools often produce multiple findings per contract, and developers typically review only a limited subset of these findings in practice [30]. For instance, in our preliminary evaluations we observed several tools emit high volumes of warnings per contract (*e.g.*, 1,485 for Smart Check and 5,205 for Solhint), which are infeasible for manual review in practice. To evaluate models according to this triage process, we compute ranked **top- K metrics** for $K \in \{1, 3, 5, 10\}$, where $K \in \{1, 3, 5, 10\}$ approximates a bounded analyst attention budget. Top- K complements micro-averaged accuracy by measuring whether correct vulnerability categories are surfaced early. Vulnerability categories are ranked using tool-provided severity or confidence signals when available, and evaluation is performed using only the top- K predicted categories per contract. This captures how effectively tools prioritize relevant vulnerability classes under practical review [47].

3.2.3 RQ3: Robustness

Detection accuracy alone does not capture the practical usability of auditing tools. We therefore explicitly measure tool robustness by tracking three execution level outcomes: **successful executions**, **execution failures**, and **empty outputs**. A successful execution occurs when a tool is able to complete the analysis and produces a vulnerability report (either correct or incorrect) for given smart contract within the dataset. Execution failures capture cases when a tool terminates (*e.g.*, due to crashes, timeouts, etc.) and is unable to

■ **Table 2** RQ1/RQ4: Micro-averaged category-level detection accuracy across SmartBugs Curated and FORGE. “Scored” ’ excludes contracts with missing/invalid outputs.

Tool	Type	SmartBugs Curated			FORGE		
		Scored	TP/FP/FN	P/R/F1	Scored	TP/FP/FN	P/R/F1
Solhint	Static (lint rules)	139	51/85/92	0.38/0.36/0.37	145	26/29/169	0.47/0.13/0.21
SmartCheck	Static (rules)	142	58/64/85	0.48/0.41/0.44	173	29/63/204	0.32/0.12/0.18
Slither	Static (IR + detectors)	140	120/267/21	0.31/0.85/0.45	142	110/283/81	0.28/0.58/0.38
Mythril	Symbolic execution	141	87/130/55	0.40/0.61/0.48	144	8/10/186	0.44/0.04/0.08
FTSMARTAUDIT	LLM	142	138/495/5	0.22/0.97/0.36	145	192/891/37	0.18/0.84/0.29
GPTSCAN	LLM	142	141/137/2	0.51/0.99/0.67	145	44/175/151	0.20/0.23/0.21
Ethainter	Logic-based (Datalog)	142	85/463/58	0.16/0.59/0.25	N/A	N/A	N/A

Note. Scored excludes contracts with missing, invalid, or non-parsable outputs.

analyze smart contracts within the dataset. Empty outputs correspond to tools that ran successfully, but did not provide any output. These indicators capture operational stability, capturing execution failure modes that affect real-world adoption beyond accuracy metrics.

3.2.4 RQ4: Cross-dataset Transfer

To assess whether conclusions from canonical benchmarks transfer to newer real-world audit-driven contracts, we compare each tool’s performance across SmartBugs Curated and FORGE. We report dataset-specific precision, recall, F1, top- K performance, and robustness outcomes, and examine changes in tool behavior across datasets. This analysis focuses on whether tools that perform well on SmartBugs maintain similar performance on FORGE.

4 Results

4.1 RQ1: Category-Level Detection Accuracy

4.1.1 Overall micro-averaged performance

Table 2 outlines the micro-averaged detection accuracy across tools on the SmartBugs Curated and FORGE benchmarks. Overall, we observed precision-recall trade-offs across tool designs: static analyzers varied between conservative and recall-oriented behavior; symbolic execution balanced precision and recall on scored runs; and LLM-based tools often achieved high recall with varying false-positive burden. Additional insights are provided below.

Static vs. LLM behavior. Across tools, the static analyzers illustrated a precision–coverage trade-off: SmartCheck and Solhint were relatively conservative (higher precision, lower recall), while Slither was a recall-oriented static baseline (high recall with substantial over-reporting). For instance, SmartCheck was conservative ($P=0.48$, $R=0.41$), Solhint was narrow-scope ($R=0.36$), and Slither acted as a high-recall static triage baseline ($R=0.85$). In contrast, the LLM-based auditors behaved as recall amplifiers. FTSMARTAUDIT achieved near-ceiling recall on SmartBugs Curated ($R=0.97$) but incurred the largest false-positive burden ($FP=495$), while GPTSCAN achieved the strongest SmartBugs Curated F1 score (0.67) due to high recall and comparatively lower false positives. These results suggest that static tools are useful for fast first-pass triage, while LLM outputs require explicit calibration through ranking, filtering, or validation to remain usable in analyst workflows.

■ **Table 3** RQ1/RQ4: Coverage effect of restricting evaluation to supported vulnerability categories.

Tool	SmartBugs Curated		FORGE		Coverage Effect Interpretation
	Supported F1	$\Delta F1$	Supported F1	$\Delta F1$	
Solhint	N/A	N/A	N/A	N/A	Not emitted
SmartCheck	0.51	+0.07	0.19	+0.01	SB coverage-limited; FG minimal
Slither	0.46	+0.01	0.38	+0.00	
Mythril	0.49	+0.01	0.08	+0.00	No change
FTSMARTAUDIT	0.36	+0.00	0.29	+0.00	No change
GPTScan	0.67	+0.00	0.21	+0.00	No change
Ethainter	0.30	+0.05	N/A	N/A	SB coverage-limited; FG TBD

Note. $\Delta F1$ = supported-category F1 – all-category F1 from Table 2. Supported categories are defined by our tool-output-to-taxonomy mapping when emitted by the evaluator. Marked N/A because its evaluator does not emit supported-category metrics.

4.1.2 Coverage-aware evaluation (supported categories only)

Because tools do not support all SmartBugs-compatible vulnerability categories equally, all-category scores can mix detection quality with category-coverage limitations. Table 3 reports the effect of restricting evaluation to supported categories only. On SmartBugs Curated, SmartCheck shows the clearest coverage effect: its recall rises from 0.41 to 0.56 and its F1 rises from 0.44 to 0.51 when evaluated only on supported categories. Ethainter also improves from 0.25 to 0.30 F1, suggesting that part of its lower all-category score reflects limited coverage. By contrast, Slither changes only slightly (0.45 to 0.46 F1), while FTSMARTAUDIT, and GPTSCAN show no change, indicating that their main limitations are not unsupported-category penalties but false positives, missed detections, or calibration. On FORGE, SmartCheck improves only slightly from 0.18 to 0.19 F1, while Slither, Mythril, FTSMARTAUDIT, and GPTSCAN show no supported-category gain. Solhint is marked N/A because its evaluator does not emit supported-category metrics.

4.1.3 Category-level detection: primary-label recall

Aggregate micro metrics can hide sharp category dependencies. Table 4 reports primary-label recall per category across all evaluated tools. These results surface practical strengths and blind spots that are not visible from aggregate micro-F1 alone.

Category-level patterns. Static rule tools have “spiky” recall. Solhint achieved strong performance on `unchecked_low_level_calls` and `time_manipulation`, but failed to detect interaction-heavy and semantic categories (`reentrancy`, `access_control`, `arithmetic`). SmartCheck was extremely strong on `unchecked_low_level_calls` but nearly blind to `reentrancy`, `bad_randomness`, and `front_running`.

Slither was the broadest non-LLM tool by recall, except for `front_running` and `short_addresses`. It hit 1.0 recall on `reentrancy`, `arithmetic`, `bad_randomness`, and `time_manipulation`, and remains high on `unchecked_low_level_calls`. Its main recall gaps were `front_running` and `short_addresses` in this protocol. Mythril was strong on `reentrancy`-like semantics but uneven elsewhere. It performed well on `reentrancy` (0.87) and `unchecked_low_level_calls` (0.88), moderate on `access_control` (0.56), but failed on `arithmetic` and `denial_of_service` under this labeling.

LLM tools behave like recall amplifiers. FTSMARTAUDIT reached 1.0 recall on nearly everything, including `short_addresses`, but did not address the ambiguous `other` bucket. GPTSCAN also showed broad SmartBugs Curated recall, reaching perfect recall on several

■ **Table 4** RQ1/RQ4: Primary-label recall by category across SmartBugs Curated and FORGE (higher is better).

Dataset	Category	#	Solhint	SmartCheck	Slither	Mythril	FTSmart	GPTScan	Ethainter
SB	unchecked_low_level_calls	51	0.90	1.00	0.96	0.88	1.00	1.00	0.98
SB	time_manipulation	5	1.00	0.20	1.00	0.40	1.00	1.00	0.00
SB	bad_randomness	8	0.00	0.00	1.00	0.25	0.88	0.88	0.00
SB	denial_of_service	6	0.00	0.33	0.83	0.00	1.00	1.00	0.00
SB	front_running	4	0.00	0.00	0.00	0.25	0.75	1.00	0.00
SB	access_control	18	0.00	0.11	0.44	0.56	1.00	1.00	0.94
SB	short_addresses	1	0.00	0.00	0.00	0.00	1.00	1.00	0.00
SB	reentrancy	30	0.00	0.00	1.00	0.87	1.00	1.00	0.00
SB	arithmetic	15	0.00	0.07	1.00	0.00	1.00	1.00	0.93
SB	other	3	0.00	0.00	0.00	0.00	0.00	0.67	1.00
FG	time_manipulation	36	0.39	0.03	0.52	0.10	0.90	0.19	N/A
FG	bad_randomness	1	0.00	0.00	0.00	0.00	0.00	0.00	N/A
FG	denial_of_service	6	0.00	0.33	0.00	0.00	0.96	0.00	N/A
FG	access_control	16	0.00	0.19	0.27	0.00	1.00	0.75	N/A
FG	reentrancy	14	0.00	0.00	0.64	0.00	0.79	0.08	N/A
FG	arithmetic	51	0.00	0.04	0.89	0.00	0.86	0.14	N/A

Note. SB = SmartBugs Curated; FG = FORGE. Solhint, Slither, and Mythril FORGE recall values are computed on the compile-success-only subset; therefore, their effective category denominators differ from the full FORGE category counts shown in the # column.

major categories while detecting some cases in the **other** bucket. **Ethainter exhibited extreme category dependence: high for explicit dataflow/permission patterns; zero for interaction/temporal vulnerabilities.**

Its near-perfect recall on **unchecked_low_level_calls**, **access_control**, and **arithmetic** suggests that Datalog rules capture certain explicit structural patterns well, while zero recall on **reentrancy**, **time_manipulation**, and **front_running** indicates missing modeling assumptions (e.g., cross-contract call ordering, temporal/transaction semantics, or ranking/heuristics). Notably, high category-level recall does not guarantee low analyst effort, as tools may still generate substantial noise even when they successfully identify the correct vulnerability family.

4.2 RQ2: Ranked Triage Effectiveness (top- K)

We evaluated ranked triage effectiveness via top- K metrics for $K \in \{1, 3, 5, 10\}$. Because many tools do not provide an explicit severity ordering, we constructed a comparable ranking by sorting predicted categories *within each contract* by the **frequency of mapped findings** for that category (most frequent first). Categories repeatedly triggered by a tool’s detectors are treated as its strongest signals for given contracts. Table 5 reports these results.

First-alert utility (K=1). First-alert utility signifies the frequency the first reported category is correct. GPTSCAN provided the strongest first-alert signal on SmartBugs Curated, with $P@1=0.887$, $R@1=0.881$, and $F1@1=0.884$. Among non-LLM tools, Slither provided the strongest first-alert signal, with $P@1=0.775$ and $F1@1=0.600$. This indicates that, although Slither’s overall micro precision is low (Table 2), the *most frequently triggered category* within each contract is often aligned with the ground truth. In practice, this corresponds to a triage-friendly behavior: analysts who only review the first suggested category are likely to see a relevant vulnerability family early.

■ **Table 5** RQ2: Ranked triage effectiveness under frequency-based top- K ordering. We report first-alert utility ($K = 1$), deep triage ($K = 10$), and recall gain $\Delta R = R@10 - R@1$.

Tool	SmartBugs Curated			FORGE		
	P/R/F1@1	P/R/F1@10	ΔR	P/R/F1@1	P/R/F1@10	ΔR
Solhint	N/A	N/A	N/A	N/A	N/A	N/A
SmartCheck	0.604/0.385/0.470	0.487/0.406/0.443	+0.021	0.30/0.08/0.13	0.32/0.12/0.18	+0.04
Slither	0.775/0.489/0.600	0.342/0.837/0.486	+0.348	0.125/0.021/0.036	0.248/0.157/0.192	+0.136
Mythril	0.542/0.275/0.364	0.420/0.606/0.496	+0.331	0.385/0.026/0.048	0.444/0.041/0.075	+0.016
FTSMARTAUDIT	0.577/0.573/0.575	0.218/0.965/0.356	+0.392	0.213/0.131/0.162	0.178/0.843/0.294	+0.712
GPTSCAN	0.887/0.881/0.884	0.507/0.986/0.670	+0.105	0.26/0.13/0.18	0.20/0.23/0.21	+0.10
Ethainter	0.372/0.357/0.364	0.155/0.594/0.246	+0.237	N/A	N/A	N/A

SmartCheck also showed strong first-alert behavior ($P@1=0.60$, $F1@1=0.47$), consistent with conservative output volume and detectors that repeatedly fire only on a limited set of patterns. Mythril’s $P@1$ was moderate (0.54), but its $R@1$ is low (0.27), suggesting that the most frequent reported category is not always the true label; correct categories may appear only after considering multiple reported categories. Among LLM-based systems, GPTSCAN showed the strongest first-alert behavior, suggesting that its dominant predicted category is often aligned with the ground truth on SmartBugs Curated. FTSMAUDIT also performed well at $K = 1$ ($P@1=0.577$, $R@1=0.573$), showing that even a highly over-reporting LLM can surface a correct primary category early when ranked by frequency. Ethainter was weakest at $K = 1$ ($P@1=0.372$, $F1@1=0.364$), indicating that its most frequent categories are often not well aligned with ground truth.

Recall gains from deeper triage. We next examine how much recall improves when analysts inspect more ranked categories, moving from $K = 1$ to $K = 10$. This shows whether relevant categories are *concentrated* among the top-ranked findings or only appear after reviewing deeper ranks. Table 5 summarizes the recall gain. Slither exhibited a large recall gain when increasing K (0.489→0.837), indicating that additional relevant categories are present but not always dominant in frequency. GPTSCAN showed a smaller recall gain on SmartBugs Curated (0.881→0.986), suggesting that most correct categories were already concentrated near the top rank. Mythril’s recall rose steadily (0.27→0.61) while maintaining relatively stable precision compared to high-recall systems, suggesting that useful categories are distributed across several findings rather than concentrated in a single dominant category. SmartCheck showed minimal recall improvement (0.38→0.41), consistent with limited output diversity: if SmartCheck misses a category, inspecting deeper ranks does not help because the tool does not emit many alternative categories per contract.

Ethainter saturated quickly: $R@1=0.36$ and $R@10=0.59$, with limited improvement beyond $K = 3$ – $K = 5$. This indicates weak ranking calibration under the frequency heuristic and aligns with its category blind spots (Table 4). For several categories Ethainter emitted few or no mapped findings, so increasing K cannot recover the missing true label. Further, while recall typically increases with K , we saw precision declined as more categories are reviewed. This decline is steepest for over-reporting tools. For example, FTSMAUDIT improved recall dramatically (0.57→0.97) but precision dropped sharply (0.58→0.22), reflecting that the long tail of additional categories is dominated by false positives. Slither’s precision also decreased from 0.78 to 0.34.

■ **Table 6** RQ3/RQ4: Dataset-scale execution robustness.

Tool	SmartBugs Curated			FORGE			Notes
	Succ.	Empty	Crash	Succ.	Empty	Crash	
Solhint	140	0	3	145	0	0	FG compile-success subset
SmartCheck	142	1	0	145	0	0	FG compile-success subset
Slither	141	0	2	142	0	3	FG analyzer failures
Mythril	100	42	1	108	0	37	FG: 36 crashes + 1 timeout
FTSMARTAUDIT	143	0	0	145	0	0	Full FG completion
GPTSCAN	143	0	0	101	44	0	FG: many empty outputs
Ethainter	138	0	5	N/A	N/A	N/A	FG not evaluated

Note. Succ. = successful non-empty output. FORGE values are reported on the 145-contract compile-success subset. Crash includes invalid outputs, analyzer crashes, and timeouts.

4.3 RQ3: Dataset-Scale Execution Robustness

Table 6 reports execution robustness outcomes. Static analyzers exhibited consistently high robustness. On SmartBugs Curated, Solhint produced usable outputs for 140/143 contracts, with three invalid outputs. SmartCheck completed on 142/143 contracts, producing one empty output and no crashes. Slither completed on 141/143 contracts, with only two invalid outputs. On FORGE, Solhint and SmartCheck produced usable outputs for all 145 contracts in the compile-success subset, while Slither produced usable outputs for 142/145 contracts and failed on three contracts. These results indicate that static analyzers generally provide reliable dataset-scale execution, even when their detection accuracy varies across categories.

Symbolic execution showed reduced robustness at dataset scale. Mythril produced usable outputs for 100/143 SmartBugs Curated contracts, with empty outputs for 42 contracts and one crash. On FORGE, Mythril produced usable outputs for 108/145 contracts, while 37 contracts resulted in crashes or timeouts. Although Mythril can provide useful accuracy on scored runs, these failures reduce its effective coverage in large-scale benchmark settings.

Among LLM-based auditors, FTSMARTAUDIT showed the strongest execution robustness, successfully producing usable outputs for all 143 SmartBugs Curated contracts and all 145 FORGE contracts. GPTSCAN also completed all SmartBugs Curated contracts, but on FORGE it produced usable outputs for only 101/145 contracts and empty outputs for 44 contracts. This suggests that GPTScan is robust on SmartBugs Curated but less reliable on FORGE, where many runs produced no usable vulnerability-category signal. Ethainter exhibited a distinct failure profile on SmartBugs Curated: it produced usable outputs for 138/143 contracts but failed with crash or invalid outcomes on five contracts, the highest crash rate among the evaluated SmartBugs tools.

4.4 RQ4: Cross-Dataset Evaluation

Table 2 compares micro-precision, micro-recall, and micro-F1 across SmartBugs Curated and FORGE. These paired results allow us to examine whether performance on the canonical SmartBugs benchmark transfers to newer audit-derived contracts. Overall, we find limited transfer: most completed tools show lower F1 on FORGE than on SmartBugs Curated, although the size of the drop varies substantially by tool (see Table 7).

The smallest cross-dataset F1 drops were observed for Slither and FTSMARTAUDIT, each decreasing by 0.07 F1. However, the two tools transferred in different ways. Slither decreased from 0.45 on SmartBugs Curated to 0.38 on FORGE, suggesting that its detector coverage and intermediate-representation-based analysis remain comparatively stable across the two

■ **Table 7** RQ4: Cross-dataset transfer from SmartBugs Curated to FORGE using micro-F1.

Tool	SB F1	FORGE F1	$\Delta F1$	Transfer Pattern
Solhint	0.37	0.21	-0.16	Partial transfer; narrow scope
SmartCheck	0.44	0.18	-0.26	Weak transfer
Slither	0.45	0.38	-0.07	Stable transfer; high FP burden
Mythril	0.48	0.08	-0.40	Poor transfer
GPTSCAN	0.67	0.21	-0.46	Large transfer drop
FTSMARTAUDIT	0.36	0.29	-0.07	Strong recall transfer; high FP burden

Note. $\Delta F1 = \text{FORGE F1} - \text{SmartBugs Curated F1}$.

benchmarks, although its FORGE precision remains low due to substantial false positives. FTSMARTAUDIT also showed a modest F1 drop from 0.36 to 0.29, primarily because its recall remained high on FORGE, but this came with a large false-positive burden.

By contrast, Mythril and GPTSCAN showed the largest transfer drops. Mythril decreased from 0.48 F1 on SmartBugs Curated to 0.08 on FORGE, indicating that its symbolic-execution strengths on canonical benchmark contracts did not generalize well to the newer FORGE subset. GPTSCAN decreased from 0.67 to 0.21 F1, showing that its strong SmartBugs Curated performance does not transfer directly to audit-derived contracts. Solhint and SmartCheck also declined on FORGE, but their results reflect narrower rule coverage and lower recall rather than complete execution failure.

Summary across RQs. Our results show that tool performance depends strongly on evaluation dimension. GPTScan leads on SmartBugs Curated accuracy and first-alert ranking, but transfers weakly to FORGE. Slither shows the strongest non-LLM transfer with broad recall, but many false positives. FTSmartAudit is highly robust and recall-oriented across datasets, but creates substantial triage burden. Mythril, SmartCheck, and Ethainter show useful strengths on specific settings, but are limited by robustness, coverage, or category selectivity. Static and LLM-based tools saw a performance drop across datasets.

5 Discussion and Future Work

Our results suggest no single type of tool dominates across accuracy, effectiveness, and robustness. We also observed substantial differences in tool performance across datasets. Based on these insights, we discuss implications for smart contract auditing in practice, benchmarking smart contract vulnerability detection tools, and future research directions.

5.1 Implications for Practice

Our findings provide insights for practitioners considering integrating smart contract vulnerability detection tools in development workflows.

Static analyzers are reliable scaffolding, but not complete auditors. Slither and SmartCheck demonstrate that modern static pipelines can execute reliably at dataset scale. However, their utility depends on analyst goals. Slither functions as a high-recall triage engine and shows the strongest cross-dataset transfer among completed FORGE tools, but it produces substantial false positives and requires downstream filtering and prioritization. SmartCheck

behaves more conservatively, achieving high precision on the categories it supports well (especially `unchecked_low_level_calls`) but failing to cover interaction-heavy vulnerabilities (e.g., `reentrancy`). Solhint is best viewed as a specialized rule checker rather than a general security auditor under this evaluation.

Symbolic execution provides strong accuracy, but suffers from dataset-scale fragility.

Mythril performs reasonably on SmartBugs Curated, but its performance transfers poorly to FORGE. On SmartBugs Curated, its frequent empty outputs (42/143, 29.4%) reveal dataset-scale fragility; on FORGE, it produced 36 crashes and one timeout on the 145-contract compile-success subset. Its FORGE F1 drops to 0.08, suggesting that symbolic-execution signals from canonical benchmarks do not generalize cleanly to newer audit-derived contracts.

LLM-based auditors behave like “recall amplifiers,” not calibrated detectors.

LLM-based tools showed strong recall, but transfer behavior differed across datasets. On SmartBugs Curated, GPTSCAN achieves the strongest micro-F1, but its FORGE F1 drops sharply to 0.21. FTSMARTAUDIT retains high FORGE recall (0.84), but with a large false-positive burden, indicating that it behaves more like a recall amplifier than a calibrated detector. The top- K results add nuance: on SmartBugs Curated, LLM tools can surface correct categories early, but on FORGE their ranking behavior is weaker and often requires deeper inspection.

Logic-based semantic analysis is promising, but uneven.

Ethainter’s results illustrate both the value and risk of declarative semantic rules: the tool can achieve near-perfect recall for some explicit pattern families (e.g., `call/permission/dataflow`-related categories) while failing completely on interaction/temporal classes (e.g., `reentrancy`, `time_manipulation`, `front_running`). Even in categories where recall is high, low precision indicates that rule sets and mappings require careful calibration and possibly additional context to avoid spurious matches. Ranking results further show that correct categories are not strongly prioritized early, limiting analyst usability.

Accuracy alone does not determine usability.

A central observation is that micro recall does not directly translate into practitioner benefit unless correct categories are surfaced early. For instance, GPTSCAN achieves the strongest first-alert behavior on SmartBugs Curated, while Slither remains a strong ranked static baseline. However, this pattern does not transfer uniformly: on FORGE, Mythril shows little recall gain with larger K , while FTSMARTAUDIT gains substantial recall only by exposing analysts to many additional false positives.

SmartCheck changes little with K , reflecting low output volume: its strengths are “early and few”, but it cannot recover missing categories by deeper inspection. Ethainter’s limited gains across K indicate weak prioritization. These differences imply that future evaluations of auditing systems should treat ranking and prioritization as first-class properties, especially for tools that generate many potential issues. In practice, teams often operate under strict time constraints [34], making $K = 1$ behavior particularly important.

5.2 Implications for Research

Our findings also suggest several directions for improving tool evaluation frameworks.

Cross-benchmark evaluations. Our cross-dataset results demonstrate that benchmark choice can substantially influence tool conclusions. For example, GPTSCAN achieved the strongest SmartBugs Curated F1 among evaluated tools, but its F1 dropped from 0.67 to 0.21 on the FORGE compile-success subset. Similarly, Mythril dropped from 0.48 to 0.08 F1, while Slither and FTSMARTAUDIT transferred more strongly, dropping only from 0.45 to 0.38 and from 0.36 to 0.29, respectively. Thus, relying on a single benchmark risks producing misleading results on tool performance, potentially leading practitioners to select tools that may not generalize to real-world contracts in production environments. Our results highlight the necessity of evaluating tools across multiple, diverse benchmarks.

Robustness metrics. Our results demonstrate robustness should be reported alongside accuracy to provide a comprehensive picture of tool reliability. For example, Mythril performed reasonably on SmartBugs Curated but showed reduced dataset-scale reliability, producing 42 empty outputs on SmartBugs and 36 crashes plus one timeout on the FORGE compile-success subset. Reporting only micro metrics can obscure this limitation. Conversely, FTSMARTAUDIT completed all FORGE runs but produced a large false-positive burden, showing that robustness must be interpreted together with accuracy and alert volume. By incorporating robustness measures into benchmarking methodologies, the research community can provide practitioners with more realistic performance expectations and help identify tools that not only achieve high accuracy but also maintain consistent, predictable behavior across smart contract vulnerability detection tasks.

Category-dependent evaluation. Aggregate metrics can mask critical blind spots. For example, our findings indicate Ethainter’s failure on reentrancy and temporal categories is not visible from overall recall alone without per-category analysis. Similarly, we observed SmartCheck achieved high performance on low-level calls, but near-zero performance on reentrancy. This suggests that practitioners should pair tools with complementary strengths rather than relying on a single auditor. Future benchmarking frameworks should also incorporate category-level results to enable more meaningful comparisons of vulnerability detection tool capabilities. In addition, the complementary strengths observed across smart contract vulnerability detection approaches motivate the needs for multi-faceted evaluations and pipelines to prevent single approach bias.

5.3 Future Work

Future work will extend the evaluation pipeline beyond category-level detection by incorporating runtime cost, alert volume, severity-aware ranking, confidence calibration, and line-level localization when available. Future benchmarks should also explore more semantically grounded normalization strategies for mapping heterogeneous tool outputs into vulnerability categories and should evaluate how developers interpret and act on tool findings in real-world workflows. Our findings motivate the potential for hybrid LLM-static approaches [1]. For example, an LLM proposes candidate smart contract vulnerability categories, and a critic/validator stage filters or confirms them using static analysis, symbolic checks, or rule-based constraints. Future work can explore hybrid auditing workflows in which LLM-based systems generate candidate vulnerability categories and static, symbolic, or logic-based analyzers validate or prioritize them. Such workflows may preserve the high recall of LLM-based tools while reducing false positives and improving robustness through selective invocation, time-bounded analysis, contract slicing, or staged analysis pipelines [38]. Future efforts are needed to investigate developer perceptions and tool effectiveness within smart contract auditing workflows. Finally, similar evaluation pipelines can compare static and LLM-based tools for other tasks.

6 Threats to Validity

Several threats should be considered when interpreting our results. First, tool behavior may be affected by implementation versions, dependencies, compiler settings, runtime limits, and LLM model configurations. We mitigate this risk by using default or recommended configurations and recording tool versions, prompts, execution scripts, and normalization logic in the replication artifact. Second, our evaluation focuses on contract-level vulnerability categories rather than line-level localization, exploitability, severity, or report understandability. This abstraction enables comparison across heterogeneous tools, but it may miss details that matter in practical audits. Third, our ground truth relies on SmartBugs annotations and FORGE audit-derived findings mapped into SmartBugs-compatible categories; labeling errors, incomplete audit reports, or imperfect CWE-to-category mappings could affect measured accuracy. Finally, our results are limited to seven Solidity/Ethereum tools and two benchmark datasets. Although SmartBugs supports comparison with prior work and FORGE adds newer audit-derived contracts, results may differ for other smart contract ecosystems, future Solidity versions, newer LLMs, or alternative prompting and orchestration strategies.

7 Conclusion

Smart contracts are critical to DeFi protocols on modern blockchain systems, yet prone to vulnerabilities. This work introduces a reproducible evaluation pipeline, and presents a benchmark of seven smart contract vulnerability detection tools across static and LLM-based auditing approaches on the SmartBugs Curated and FORGE datasets. Our results show none of the sampled tools simultaneously achieve high detection accuracy, low false-positive burden, and robust dataset-scale execution – motivating the need for compositional and hybrid pipelines that combine the complementary strengths of varied approaches. Based on our findings, we provide implications for practitioners and researchers, in addition to future research directions to establish a rigorous benchmarking foundation for evaluating next-generation smart contract auditing systems.

8 Data Availability Statement

All data and study artifacts, including our evaluation pipeline scripts, prompt templates, etc., are available to support reproducibility and reuse.¹ SmartBugs Curated² is made available by Durieux et al. FORGE³ is made available by Chen et al.

References

- 1 Vaibhav Agrawal and Kiarash Ahi. Llm-driven sast-genius: A hybrid static analysis framework for comprehensive and actionable security. *arXiv preprint*, 2025. doi:10.48550/arXiv.2509.15433.
- 2 Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts. In *Principles of Security and Trust (POST)*. Springer, 2017.
- 3 Sebastian Baltes et al. Guidelines for empirical studies in software engineering involving large language models, 2026. arXiv:2508.15503.

¹ <https://github.com/shawalkhalid667/smartcontracts-auditor-benchmark>

² <https://github.com/smartbugs/smartbugs-curated>

³ <https://github.com/shenyimings/FORGE-Artifacts>

- 4 Lexi Brent, Neville Grech, Sifis Lagouvardos, et al. Ethainter: A smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '20, pages 16–31. ACM, 2020. doi:10.1145/3385412.3385990.
- 5 Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, et al. Smart contract and defi security tools: Do they meet the needs of practitioners? *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 716–728, 2023. doi:10.1145/3597503.3623302.
- 6 Jiachi Chen, Yiming Shen, Jiashuo Zhang, et al. Forge: An llm-driven framework for large-scale smart contract vulnerability dataset construction. *arXiv preprint*, 2025. arXiv:2506.18795.
- 7 Ra-Yeon Choi, Dong-Hyuk Im, Yeji Song, et al. Smart contract vulnerability detection using large language models and graph structural analysis. *Computers, Materials & Continua*, 83(1):785–801, 2025. doi:10.32604/cmc.2025.061185.
- 8 Hanting Chu et al. A survey on smart contract vulnerabilities: Data sources, detection and repair. *Inf. Softw. Technol.*, 159:107221, 2023. doi:10.1016/j.infsof.2023.107221.
- 9 Consensys. Mythril: Security analysis tool for ethereum smart contracts. <https://github.com/ConsenSys/mythril>, 2018.
- 10 Hao Ding et al. Smartguard: An llm-enhanced framework for smart contract vulnerability detection. *Expert Syst. Appl.*, 269:126479, 2025. doi:10.1016/j.eswa.2025.126479.
- 11 Thomas Durieux et al. Smartbugs: A framework to analyze solidity smart contracts. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2020.
- 12 Thomas Durieux, J. Ferreira, Rui Abreu, et al. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. *International Conference on Software Engineering (ICSE)*, pages 530–541, 2019. doi:10.1145/3377811.3380364.
- 13 Thomas Durieux, Joao Ferreira, et al. Empirical review of automated smart contract security tools. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2020.
- 14 Phan The Duy, Nghi Hoang Khoa, Nguyen Huu Quyen, et al. Vulnsense: Efficient vulnerability detection in ethereum smart contracts by multimodal learning with graph neural network and language model, 2023. doi:10.48550/arXiv.2309.08474.
- 15 Josselin Feist, Gustavo Grieco, et al. Slither: A static analysis framework for smart contracts. In *International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, 2019.
- 16 Zhipeng Gao, Vinoj Jayasundara, Lingxiao Jiang, et al. Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 394–397. IEEE, 2019.
- 17 Asem Ghaleb and K. Pattabiraman. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020. doi:10.1145/3395363.3397385.
- 18 Sihao Hu, Tiansheng Huang, Fatih Ilhan, et al. Large language model-powered smart contract vulnerability detection: New perspectives. *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, pages 297–306, 2023. doi:10.1109/tps-isa58951.2023.00044.
- 19 Peter Ince, Jiangshan Yu, Joseph K. Liu, et al. Generative large language model usage in smart contract vulnerability detection. *ArXiv*, abs/2504.04685, 2025. doi:10.48550/arxiv.2504.04685.
- 20 Jun Kevin and Pujianto Yugopuspito. Smartllm: Smart contract auditing using custom generative ai. In *2025 International Conference on Computer Sciences, Engineering, and Technology Innovation (ICoCSETI)*, pages 260–265. IEEE, 2025.
- 21 Shawal Khalid and Chris Brown. Software engineering approaches adopted by blockchain developers. In *2023 Tenth International Conference on Software Defined Systems (SDS)*, pages 1–6. IEEE, 2023. doi:10.1109/SDS59856.2023.10329007.

- 22 Shawal Khalid and Chris Brown. Evaluating capabilities and perspectives of generative ai tools in smart contract development. In *Proceedings of the 7th ACM International Symposium on Blockchain and Secure Critical Infrastructure*, pages 1–12, 2025.
- 23 Shawal Khalid, Huayu Liang, and Chris Brown. Exploring user perceptions of crypto signals: An empirical study from social media. In *2024 IEEE International Conference on Blockchain (Blockchain)*, pages 483–490. IEEE, 2024. doi:10.1109/BLOCKCHAIN62396.2024.00071.
- 24 Shawal Khalid, Huayu Liang, and Chris Brown. Decoding the influence: Analyzing the impact of crypto signals on software repositories. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering-Companion (SANER-C)*, pages 124–131. IEEE, 2025. doi:10.1109/SANER-C66551.2025.00025.
- 25 Kaixuan Li, Yue Xue, Sen Chen, et al. Static application security testing (sast) tools for smart contracts: How far are we? *Proceedings of the ACM on Software Engineering*, 1:1447–1470, 2024. doi:10.1145/3660772.
- 26 Loi Luu, Duc-Hiep Chu, Hrishi Olickel, et al. Making smart contracts smarter. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.
- 27 Wei Ma, Daoyuan Wu, Yuqiang Sun, et al. Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications. *arXiv preprint*, 2024. arXiv:2403.16073.
- 28 Mark Mossberg, Felipe Manzano, Eric Hennenfent, et al. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.
- 29 Virraji Mothukuri, Reza M Parizi, and James L Massa. Llmsmartsec: Smart contract security auditing with llm and annotated control flow graph. In *2024 IEEE International Conference on Blockchain (Blockchain)*, pages 434–441. IEEE, 2024. doi:10.1109/BLOCKCHAIN62396.2024.00064.
- 30 Kien-Tuan Ngo, Dinh-Truong Do, Thu-Trang Nguyen, et al. Ranking warnings of static analysis tools using representation learning. In *Proceedings of the 28th Asia-Pacific Software Engineering Conference (APSEC)*, pages 327–337. IEEE, 2021. doi:10.1109/APSEC53868.2021.00040.
- 31 Ivica Nikolić, Aashish Kolluri, Ilya Sergey, et al. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th annual computer security applications conference*, pages 653–663, 2018.
- 32 Nonso Okika, Omoshalewa A. Adeosun, Oluwatobi J. Ogunjide, et al. Smart contract vulnerability in defi: Assessing security risk in blockchain-based lending platforms. *Global Journal of Engineering and Technology Advances*, 2025. doi:10.30574/gjeta.2025.22.3.0064.
- 33 Nathaniel Popper. A hacking of more than \$50 million dashes hopes in the world of virtual currency. *New York Times*, 2016. URL: <https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have-removed-more-than-50-million-from-experimental-cybercurrency-project.html>.
- 34 Meng Ren, Fuchen Ma, et al. Making smart contract development more secure and easier. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering, ESEC/FSE 2021*, pages 1360–1370, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3468264.3473929.
- 35 Meng Ren, Zijing Yin, et al. Empirical evaluation of smart contract testing: what is the best choice? In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2021*, pages 566–579, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3460319.3464837.
- 36 Niklas Risse, Jing Liu, and Marcel Böhme. Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):388–410, 2025. doi:10.1145/3728887.
- 37 Francesco Salzano, Cosmo K. Antenucci, Simone Scalabrino, et al. An empirical analysis of vulnerability detection tools for solidity smart contracts using line level manually annotated vulnerabilities. *ArXiv*, abs/2505.15756, 2025. doi:10.48550/arxiv.2505.15756.

- 38 Ze Sheng et al. Llms in software security: A survey of vulnerability detection techniques and insights. *ACM Comput. Surv.*, 58(5), November 2025. doi:10.1145/3769082.
- 39 Jingjing Song, Guangquan Xu, Yidong Li, and Chunhua Su. Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering*, 8:1133–1144, 2020. doi:10.1109/tnse.2020.2968505.
- 40 Yuqiang Sun, Daoyuan Wu, Yue Xue, et al. 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, ICSE '24*, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3597503.3639117.
- 41 Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, et al. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018.
- 42 Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 103–119. IEEE, 2021. doi:10.1109/EUROSP51992.2021.00018.
- 43 Petar Tsankov et al. Securify: Practical security analysis of smart contracts. In *ACM SIGSAC conference on computer and communications security*, pages 67–82, 2018.
- 44 Nuno Veloso. Conkas: A modular and static analysis tool for ethereum bytecode, 2023.
- 45 F. Vidal, N. Ivaki, et al. Vulnerability detection techniques for smart contracts: A systematic literature review. *J. Syst. Softw.*, 217:112160, 2024. doi:10.1016/j.jss.2024.112160.
- 46 Yichen Wang, Xiangfu Zhao, Long He, et al. Contractgnn: Ethereum smart contract vulnerability detection based on vulnerability sub-graphs and graph neural networks. *IEEE Transactions on Network Science and Engineering*, 2024.
- 47 Zhiyuan Wei, Jing Sun, Zijian Zhang, et al. Llm-smartaudit: Advanced smart contract vulnerability detection. *ArXiv*, abs/2410.09381, 2024. doi:10.48550/arxiv.2410.09381.
- 48 Zhiyuan Wei, Jing Sun, Zijian Zhang, et al. Ftsmartaudit: A knowledge distillation-enhanced framework for automated smart contract auditing using fine-tuned llms, 2025. arXiv:2410.13918.
- 49 Lei Yu et al. Smart-llama: two-stage post-training of large language models for smart contract vulnerability detection and explanation. *arXiv preprint*, 2024. arXiv:2411.06221.
- 50 William Zhang, Sebastian Banescu, Leodardo Pasos, et al. Mpro: Combining static and symbolic analysis for scalable testing of smart contract. *IEEE International Symposium on Software Reliability Engineering*, pages 456–462, 2019. doi:10.1109/issre.2019.00052.
- 51 Haozhe Zhou et al. The state of ethereum smart contracts security: Vulnerabilities, counter-measures, and tool support. *J. Cybersecur. Priv.*, 2:358–378, 2022. doi:10.3390/jcp2020019.