Why Do Smart Contract Security Tools Fall Short?

Anonymous Author(s)

Abstract

Smart contracts have become integral to high-value ecosystems such as decentralized finance (DeFi), yet their security remains a persistent challenge. Despite the availability of automated analyzers for detecting vulnerabilities, these tools often exhibit inconsistent performance and struggle to gain developer trust.

This paper presents a comprehensive study bridging technical evaluation with developer perspectives on smart contract security tools. We empirically benchmark five widely-used analyzers (i.e., Confuzzius, Mythril, Osiris, Oyente, and Slither) across 653 real-world smart contracts containing known vulnerabilities, focusing on reentrancy, suicide, and integer arithmetic errors. Our results reveal significant variability in detection accuracy, high false-positive rates, and long analysis times. Complementing this evaluation, we surveyed 150 smart contract developers and auditors, identifying key trust barriers: false positives, unclear explanations, and a lack of actionable guidance.

By uniting quantitative benchmarking with developer insights, our study highlights the urgent need for security analyzers that balance technical precision with usability and developer-oriented reporting. We provide actionable recommendations aimed at improving tool effectiveness and fostering wider adoption, ultimately contributing to more secure blockchain ecosystems.

CCS Concepts

• **Software and its engineering → Software verification and validation**.

Keywords

Blockchain Security, Vulnerability Detection, Empirical Evaluation, Developer Survey, Analyzers Limitations

ACM Reference Format:

Anonymous Author(s). 2026. Why Do Smart Contract Security Tools Fall Short?. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE '26).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Blockchain platforms (e.g., Ethereum [6]) have popularized smart contracts as a foundational technology for decentralized applications. To enable transparent and verifiable transactions, critical ecosystems such as decentralized finance (DeFi) [47] and non-fungible token (NFT) marketplaces [46] rely on these self-executing programs. This technological shift has fueled rapid market growth, with the global blockchain industry valued at US$ 20.1 billion in 2024 and projected to reach US$ 248.9 billion by 2029 [30]. However, smart contracts are also frequent targets for security exploits. In 2024 alone, 303 attacks led to cryptocurrency losses totaling US$2.2 billion, a 21% increase from 2023 [9], reflecting both the scale of adoption and its security challenges. Well-known incidents, from the 2016 DAO hack [43] to recent reentrancy and flash loan attacks [48], highlight recurring vulnerability patterns.

To mitigate these risks, automated security analyzers have become essential for auditing smart contracts at scale. Existing tools leverage three main analysis techniques, each with trade-offs in terms of precision, scalability, and coverage. First, static analysis inspects code without execution, offering fast, early vulnerability detection (e.g., Vandal [5] and Slither [19]). Second, symbolic execution explores program paths using symbolic inputs, uncovering deep logic flaws (e.g., Oyente [29], Mythril [32], Osiris [45], and Manticore [31]). Finally, fuzzing generates random or systematic inputs to trigger vulnerabilities during execution (e.g., Confuzzius [44] and Echidna [23]). Despite their potential, these tools exhibit inconsistent detection rates for critical vulnerability classes, such as reentrancy [41], suicide [42], and integer arithmetic errors [40]. These limitations are compounded by high false-positive rates, prolonged analysis times, and usability shortcomings[17]. Developers in other domains and programming languages frequently cite a lack of actionable explanations and report low trust in analyzer outputs [16, 27], leading to ignored warnings and unaddressed vulnerabilities. It remains unclear whether smart contract developers face the same challenges.

To systematically investigate the gap between the performance of smart contract analyzers and their practical adoption, we have conducted an empirical evaluation of five widely-used analyzers (i.e., Confuzzius, Mythril, Osiris, Oyente, and Slither) on 653 real-world smart contracts with known vulnerabilities, quantifying detection effectiveness and efficiency. We have also surveyed 150 smart contract developers, capturing their experiences, trust factors, and expectations regarding vulnerability reporting. Our work has been primarily guided by the following research questions:

- **RQ1**: How effective are current security analyzers in detecting security vulnerabilities in smart contracts?
- **RQ2**: What are the trade-offs between detection accuracy and computational efficiency for each security analyzer?
- **RQ3**: What are the most common security vulnerabilities cited by developers?
- **RQ4**: What is an acceptable runtime performance of a security analyzer according to developers?
- **RQ5**: What are the factors that affect developer trust/confidence in security analyzers?
- **RQ6**: Why do developers ignore reported vulnerabilities by security analyzers?

ICSE '26, Rio De Janeiro, Brazil

ACM ISBN 978-1-4503-XXXX-X/26/04
https://doi.org/XXXXXXX.XXXXXXX

- **RQ7**: What types of explanations for vulnerabilities flagged by security analyzers do developers find most effective in enhancing trust and facilitating remediation?

While prior studies [10, 17, 38] have evaluated vulnerability tools or documented attacks, none provide an integrated view of tool effectiveness and developer trust. For example, Durieux et al. [17] study tool precision across a large corpus but do not consider real-world exploits or developer perspectives. Perez and Livshits [38] analyze flagged vulnerabilities versus exploitation rates but do not assess tool performance directly. Chaliasos et al. [10] combine tool benchmarking with a developer survey, yet do not explicitly link tool performance characteristics with trust and usability factors.

Our work bridges this gap by combining large-scale empirical evaluation with a developer-centric trust study. We provide the first end-to-end analysis of how technical limitations translate into reduced confidence, offering concrete guidance for improving smart contract security tooling. In particular, we make the following contributions:

- A rigorous assessment of the strengths and weaknesses of current security analyzers, revealing significant variability in detection accuracy and computational efficiency across vulnerability types.
- A detailed understanding of developer experiences, identifying key factors (e.g., false positives and explanation quality) that influence trust and the decision to act upon reported vulnerabilities.
- Evidence-based recommendations for improving security analyzers, aligning tool capabilities with developer needs to enhance usability and effectiveness.

Availability To promote transparency and facilitate further research, all data and analysis from this study—including the dataset of 653 smart contracts, benchmarking results, and anonymized survey questions and responses—are publicly available in our replication package at [2].

2 Background

2.1 Smart Contracts

Smart contracts are self-executing programs deployed on blockchain networks (e.g., Ethereum), enforcing agreements without intermediaries [7]. They are written in languages such as Solidity and execute automatically based on predefined conditions, with their code and execution history immutably stored on the blockchain. Smart contracts power decentralized applications (DApps) (e.g., decentralized finance (DeFi) [47] and non-fungible token (NFT) marketplaces [46]). However, their immutability means that any vulnerabilities present at deployment cannot be easily fixed, necessitating advanced security tools to protect against exploitation.

2.2 Smart Contract Vulnerabilities

Smart contract vulnerabilities expose deployed contracts to exploitation, resulting in financial losses or unintended behavior [3, 48]. Common issues include reentrancy, which enables recursive calls to drain funds; suicide attacks, allowing unauthorized contract destruction; and integer overflow/underflow, causing erroneous calculations. These vulnerabilities are typically exploited through malicious transactions that manipulate contract logic. Detecting these transactions in real-time (i.e., while they are still in the `mempool`) is essential for preventing exploitation before the transactions are confirmed on the blockchain.

```
contract DAOContract {
  uint256 public balance;
  function withdraw(uint256 _amount) public {
    require(balance >= _amount);
    msg.sender.call{value: _amount}("");
    balance -= _amount; }
}
contract AttackerContract {
  DAOContract public target;
  receive() external payable {
    if (target.balance > 0) {
    target.withdraw(address(this).balance);
    } }
}
```

Figure 1: An example illustrating reentrancy.

```
contract WalletLibrary {
  function kill() public {
    selfdestruct(msg.sender); }
}
contract MyWallet {
  WalletLibrary public library;
  constructor(address _libraryAddress) public{ library =
        WalletLibrary(_libraryAddress); }
  // Other functions using the library
}
```

Figure 2: An example illustrating suicide attack.

2.2.1 Reentrancy. This vulnerability occurs when an external contract recursively calls a function in the original contract before its state is updated, potentially leading to unauthorized fund withdrawals or state manipulation [41]. The 2016 DAO hack exploited this flaw, resulting in the theft of 3.6 million ETH (approximately US$60 million at the time), which required a controversial Ethereum hard fork to recover funds [43]. Figure 1 demonstrates a vulnerable contract that sends Ether to `msg.sender` before updating its balance, allowing an attacker to re-enter `withdraw()` repeatedly, draining funds. Mitigation involves updating state variables before external calls or using reentrancy guards (e.g., `ReentrancyGuard` from OpenZeppelin [36]).

2.2.2 Suicide Attacks. They are also known as self-destruct vulnerabilities. These attacks occur when an attacker exploits inadequate access controls to invoke a contract's `selfdestruct()` function, permanently deleting the contract and transferring its remaining Ether to a designated address. A notable incident involved the Parity Multisig Wallet vulnerability in 2017, where an attacker exploited an unprotected library contract to gain ownership and call `selfdestruct()`, as we show in Figure 2. This attack destroyed the library contract, freezing approximately 500,000 ETH (valued at US$150 million) in dependent wallets [42]. The incident highlights

```
pragma solidity ^0.6.0;
contract VulnerableToken {
    mapping(address => uint8) public balances;
    uint8 public totalSupply;
    function transfer(address to, uint8 amount) public {
        require(balances[msg.sender] >= amount, "Insufficient
            balance");
        balances[msg.sender] -= amount;
        balances[to] += amount;
        totalSupply += amount;
    }
}
```

Figure 3: An example illustrating integer overflow.

the critical need for securing privileged functions with robust access control mechanisms, such as ownership checks or function modifiers.

2.2.3 Integer Overflow/Underflow. These vulnerabilities, prevalent in Solidity versions prior to 0.8.0, occur when arithmetic operations exceed a variable's data type limits, resulting in unintended values [40]. These issues allow attackers to manipulate contract logic (e.g., create unauthorized tokens or bypass balance checks). Figure 3 illustrates a vulnerable contract where `totalSupply` or `balances[to]` may overflow or underflow during arithmetic operations, enabling exploitation. For instance, an attacker could reduce `balances[msg.sender]` below zero, triggering an underflow to a large positive value, or increase `totalSupply` beyond its intended limit, causing an overflow. Since Solidity 0.8.0, built-in overflow and underflow checks revert transactions on arithmetic errors, mitigating these risks. For earlier versions, developers should use libraries such as OpenZeppelin's `SafeMath` [36] to enforce safe arithmetic operations.

3 Empirical Evaluation of Security Analyzers

To assess the effectiveness of current security analyzers in identifying actual vulnerabilities in smart contracts, we have conducted an empirical experiment that focus on the vulnerability types that we have previously discussed.

3.1 Dataset Collection

We collected a total of 717 smart contract addresses along with their vulnerability labels from the existing literature [1, 11, 22, 25], as well as from contracts audited by reputable security firms (e.g., Trail of Bits [34], ConsenSys Diligence [15], OpenZeppelin [35]).

To provide the ground truth for our evaluation, we use the labels that the authors of the cited literature and auditing firms have manually assigned. To prepare the dataset, we first retrieved the source code for these contracts from Etherscan [18]. We then apply a cleaning process to the Solidity code, which involves removing comments, correcting missing pragma versions, and resolving other minor issues to ensure compilability. After this process, we retained 653 compilable contracts for analysis.

To validate the integrity of our dataset, we systematically verified its adherence to the principle that *vulnerability does not imply exploitation*, as established by Perez and Livshits [38]. This principle recognizes that a smart contract's susceptibility to known vulnerabilities does not necessarily result in exploitation due to factors such as insufficient attacker incentives, technical barriers, or exploit complexity. To ensure our dataset of 653 smart contracts reflects this reality, we conducted a rigorous inspection of each contract's source code, retrieved from Etherscan [18], to classify contracts as exploitable or non-exploitable. We performed a manual code analysis to identify vulnerability patterns, followed by a transaction history review to detect exploitation evidence. Specifically, for reentrancy vulnerabilities, we examined whether functions used the `call()` method to transfer Ether to external contracts before updating state variables, a pattern enabling recursive calls. For suicide vulnerabilities, we checked for unauthorized `selfdestruct` invocations and reviewed transaction logs for evidence of contract destruction. For integer overflow/underflow, we identified arithmetic operations that could exceed a variable type's maximum or minimum bounds without using secure libraries like SafeMath [36]. These findings were cross-referenced with reported incidents from security audits and literature [1, 11, 22, 25] to confirm exploitation status. This meticulous curation enables our empirical analysis to capture the full spectrum of smart contract security, supporting the development of tools that not only detect vulnerabilities but also assess their practical exploitability.

Table 1: Distribution of contracts in our dataset.

Vulnerability	Safe	Vulnerable	Total
Reentrancy	90	95	185
Suicide Attacks	89	220	309
Integer Overflow/Underflow	96	63	159
Total	275	378	653

Our dataset comprises contracts exhibiting at least one of three specific vulnerability types: reentrancy, suicide attacks, and integer overflow/underflow. Table 1 presents the distribution of safe and vulnerable contracts for each vulnerability type. This diverse dataset enables a comprehensive assessment of security analyzers across common smart contract vulnerabilities.

3.2 Security Analyzers Selection

To select our subject analyzers, we initially considered a pool of 133 tools from a recent survey [26]. To ensure relevance and reliability, our primary selection criteria were: open-source availability for transparency, active maintenance for ongoing reliability, popularity among practitioners for community trust, and frequent academic citations for validation. To ensure methodological diversity, we included tools that employ static analysis, symbolic execution, and fuzzing. This process resulted in five tools:

- Confuzzius [44] is a hybrid fuzzing tool for Ethereum smart contracts that combines data-dependency analysis with traditional fuzz testing. It uses taint tracking and lightweight symbolic execution to guide input generation towards potentially vulnerable paths, improving detection of complex issues such as reentrancy and arithmetic bugs. Confuzzius balances fuzzing scalability with deeper bug coverage by focusing on dependency-aware input selection.

- Mythril (v0.24.7) [32] is a widely adopted symbolic execution engine for analyzing Ethereum Virtual Machine (EVM) bytecode. Mythril systematically explores execution paths with symbolic transaction parameters, builds control-flow graphs, and uses the Z3 SMT solver [14] to identify vulnerabilities such as reentrancy and unchecked call returns. It produces concrete exploit traces, offering actionable outputs alongside vulnerability reports.
- Osiris [45] is a symbolic execution–based tool that detects integer-related vulnerabilities (e.g., overflows and underflows). Osiris targets arithmetic-specific invariants through focused SMT queries, offering precise detection even in complex control flows (e.g., loop-heavy contracts).
- Oyente [29] is one of the first symbolic execution frameworks for Ethereum smart contracts. Oyente detects issues such as reentrancy, transaction-ordering dependence, and unchecked call returns. It uses path feasibility checks to reduce false positives and can generate concrete test inputs. Oyente's modular design laid the foundation for subsequent symbolic execution tools and continues to be influential in both academic research and security audits.
- Slither (v0.10.4) [19] is a fast static analyzer that operates on Solidity source code. Slither constructs SlitherIR, its own Intermediate Representation (IR), and then applies more than 70 vulnerability and quality checks (e.g., reentrancy, integer bugs, improperly enforced access controls). Its lightweight performance enables analyzing contracts in 2–3 seconds, making it practical for large-scale detection and integration into Continuous Integration (CI) pipelines.

3.3 Benchmarking Results

To conduct our empirical evaluation, we ran each tool on the entire dataset for the vulnerability types it supports. We then compared their outputs against ground truth labels to assess their performance, with a timeout of 900 seconds. The vulnerability types include reentrancy, suicide attacks, and integer overflow/underflow. Our performance metrics are: precision (P), recall (R), F1-score (F1), false negative rate (FNR), false positive rate (FPR), and average analysis time per contract. These metrics provide insights into each tool's accuracy, ability to identify vulnerabilities, and computational efficiency. Tables 2 to 4 show the results for each vulnerability type. Each table lists the evaluated tools, their performance metrics, and the average analysis time per contract.

3.3.1 Reentrancy. Confuzzius demonstrates superior performance with an F1-score of 94.6%, driven by high precision (94.8%) and recall (94.6%), reflecting its hybrid fuzzing approach's effectiveness in identifying reentrancy vulnerabilities. Slither follows closely with an F1-score of 90.7% and the lowest FNR (1.1%), indicating minimal missed vulnerabilities, though its FPR (17.8%) suggests occasional overflagging. Oyente and Osiris achieve F1-scores of 85.2% and 84%, respectively, but their higher FPRs (26.7% and 28.9%) indicate a tendency to flag safe contracts as vulnerable. Mythril significantly underperforms with an F1-score of 49% and an FNR of 80%, missing most vulnerable contracts. In terms of efficiency, Slither completes analysis in 1.33 seconds, far outperforming Confuzzius (565.12 seconds) and Mythril (723.66 seconds).

Table 2: Results for reentrancy.

Tool	P	R	F1	FNR	FPR	Time (s)
Confuzzius	94.8%	94.6%	94.6%	2.1%	8.9%	565.12
Mythril	64.6%	55.7%	49.0%	80.0%	6.7%	723.66
Osiris	86.5%	84.3%	84.0%	3.2%	28.9%	117.25
Oyente	87.3%	85.4%	85.2%	3.2%	26.7%	7.91
Slither	91.9%	90.8%	90.7%	1.1%	17.8%	1.33

Table 3: Results for suicide attacks.

Tool	P	R	F1	FNR	FPR	Time (s)
Confuzzius	78.1%	51.1%	50.3%	66.8%	4.5%	481.01
Mythril	77.5%	49.2%	47.8%	69.5%	4.5%	491.71
Slither	70.0%	57.6%	59.4%	49.1%	25.8%	1.71

Table 4: Results for integer overflow/underflow.

Tool	P	R	F1	FNR	FPR	Time (s)
Confuzzius	47.5%	55.3%	47.9%	12.7%	14.6%	304.38
Mythril	77.8%	64.8%	54.7%	88.9%	0.0%	664.07
Osiris	84.1%	83.0%	83.2%	12.7%	19.8%	240.04
Oyente	75.7%	75.5%	75.6%	28.6%	21.9%	17.61

3.3.2 Suicide Attacks. Performance across all tools is notably weaker for suicide attacks. Slither leads with an F1-score of 59.4%, followed by Confuzzius (50.3%) and Mythril (47.8%). High FNRs plague all tools, with Mythril and Confuzzius missing 69.5% and 66.8% of vulnerable contracts, respectively, indicating challenges in detecting suicide attacks. Slither has a lower FNR (49.1%), but its FPR (25.8%) suggests more false positives. Slither remains the fastest at 1.71 seconds, while Confuzzius and Mythril require over 480 seconds per contract, highlighting their computational intensity.

3.3.3 Integer Overflow/Underflow. Osiris excels with an F1-score of 83.2%, leveraging its specialized symbolic execution for integer-related vulnerabilities, with a low FNR (12.7%) and FPR (19.8%). Oyente follows with an F1-score of 75.6%, while Mythril and Confuzzius lag at 54.7% and 47.9%, respectively, with FNRs (88.9% and 12.7%), indicating poor detection of vulnerable contracts. Oyente is the fastest at 17.61 seconds per contract, followed by Osiris (240.04 seconds), and Confuzzius and Mythril being much slower.

3.4 Discussion

The benchmarking results reveal distinct strengths and limitations among the security analyzers across the vulnerability types that we have studied.

Confuzzius excels in detecting reentrancy vulnerabilities due to its hybrid fuzzing and data-dependency analysis, achieving near-perfect F1-score. However, its low F1-score for integer overflow/underflow (47.9%) and suicide (50.3%) highlight its limited generalization across vulnerability types. Its long analysis time (e.g., 565.12 seconds for reentrancy) restricts scalability for large contract sets.

Slither provides a robust balance of high accuracy, with F1-scores of 90.7% for reentrancy and 59.4% for suicide attacks, as well as exceptional speed (around 2 seconds per contract). However, it does not support detecting integer overflow/underflow. Although its low FNR for reentrancy indicates its reliability for detecting some critical vulnerabilities, its FPR (17.8%) suggests over-flagging which indicates an increase on the cost of false alarms.

Since Osiris is specialized in integer-related vulnerabilities, it scored the best F1-score (83.2%) for integer overflow/underflow. However, its relatively moderate F1-score for reentrancy (84%) and not detecting suicide attacks indicate a narrower focus. Additionally, its relatively long analysis time remains a bottleneck.

Mythril has suboptimal F1-scores (49% for reentrancy, 47.8% for suicide attacks, and 54.7% for integer overflow/underflow) and high FNRs (notably 80% for reentrancy). Additionally, its long analysis times (e.g., 723.66 seconds for reentrancy) limits its practical effectiveness for scanning large volumes of smart contracts, whether for assessing the blockchain's current state or monitoring new contracts in real time.

Oyente offers consistent relative F1-scores across its supported vulnerabilities (85.2% for reentrancy and 75.6% for integer overflow/underflow), with moderate analysis times (e.g., 7.91 seconds for reentrancy). However, its high FPRs (e.g., 26.7% for reentrancy) suggest a need for improved filtering of false positives.

RQ1 and **RQ2**:

- No tool excels across all vulnerabilities.
- High FNRs lead to missing critical vulnerabilities, while high FPRs increase manual review burden.
- Detection capabilities show significant variability, with suicide attacks proving the most challenging to detect (F1-scores below 60%).
- Slither and Oyente have the best running time, while Confuzzius and Mythril are the slowest.

3.5 Call to Action

Our empirical evaluation highlights concrete limitations in existing smart contract security analyzers, including inconsistent detection rates, high false-positive counts, and inefficient runtimes. To address these challenges, we propose the following targeted recommendations, each grounded in our findings:

(1) *Integrate Hybrid Analysis for Broader Coverage.* Confuzzius has the highest detection rates for reentrancy, reflecting the effectiveness of combining fuzzing with symbolic execution. Extending this hybrid analysis approach to symbolic execution tools (e.g., Mythril) would reduce false negatives by improving path exploration in complex contracts.
(2) *Optimize Symbolic Execution with Bounded Exploration.* Mythril has high FNRs and runs the slowest, likely due to its exhaustive symbolic execution. Techniques such as bounded symbolic execution [8], feasibility checks (e.g., as implemented in Oyente), and SMT query optimization [13] would improve both efficiency and vulnerability coverage.
(3) *Expand Specialized Vulnerability Patterns.* Thanks to its specialized arithmetic predicates, Osiris has the best performance in detecting integer overflow/underflow vulnerabilities. Incorporating similar tailored vulnerability patterns into other tools (e.g., Slither and Confuzzius) would enhance their detection capabilities for integer overflow/underflow and suicide attacks.
(4) *Reduce False Positives via Cross-Validation.* High FPRs observed in Osiris and Oyente undermine trust in their outputs. Integrating static analysis results from other tools (e.g., Slither) as a cross-validation step would help filter implausible vulnerability reports while maintaining acceptable analysis times.
(5) *Leverage Parallelization for Scalability.* Confuzzius and Mythril have substantially longer analysis times compared to other tools. Parallelizing symbolic execution and fuzzing workloads, as well as incorporating static pre-analyses to prioritize high-risk paths, could mitigate this limited scalability.
(6) *Develop Ensemble-Based Analyzers.* No single tool consistently outperformed others across all vulnerability types and performance metrics. Developing ensemble frameworks that combine outputs from multiple analyzers (perhaps in a game-theoretic fashion [24]) would leverage each tool's strengths, providing broader coverage and reducing the impact of individual tool weaknesses.

By directly linking these recommendations to the limitations that we have observed for each tool, we aim to provide actionable priorities for developing smart contract analyzers that balance detection accuracy, runtime efficiency, and developer trust.

4 Surveying Smart Contract Developers

To investigate the relationship between the performance of security analyzers and developer trust, we conducted a comprehensive survey in early 2025 targeting experienced blockchain practitioners. Our study aims at quantifying how usability issues (e.g., high FPRs and insufficient contextual explanations) undermine confidence in automated vulnerability reports. The survey, approved by the Institutional Review Board (IRB) at ***ANONYMOUS***, explored developers' experiences, preferences, and challenges when using security analyzers, with a particular focus on clarity, relevance, and usefulness of the reported vulnerabilities.

4.1 Survey Design

We designed the survey to take approximately 25–30 minutes to complete. To ensure participant privacy and encourage candid responses, we administered it anonymously via Prolific [39]. We targeted individuals actively involved in smart contract development, including roles such as smart contract developers, security auditors, quality assurance testers, technical architects, and researchers. We ended up with a diverse sample of 150 respondents with varying levels of expertise.

The survey contains 25 questions that address the remaining research questions (**RQ3**–**RQ7**), by covering roles, experience, tool usage, confidence in security analyzers, reasons for ignoring reported vulnerabilities, and preferences for vulnerability explanations. Questions were a mix of multiple-choice, checkbox, and open-ended formats to capture both quantitative metrics and qualitative insights. We have made the complete questionnaire and anonymized participant responses publicly available [2]. The questionnaire consists of the following main sections:

- **Participant Background (Q1–Q6)**: These questions gathered demographic and contextual data, including respondents' roles, years of experience in smart contract development, preferred programming languages (e.g., Solidity, Vyper, Rust), types of projects (e.g., DeFi, NFTs, enterprise solutions), the importance of security in their workflows, and most common vulnerabilities (**RQ3**).
- **Usage of Security Analyzers (Q7–Q15)**: This section explored how and when developers use security analyzers, covering tools employed, frequency of use, development stages (e.g., coding, testing, auditing), reasons for use (e.g., vulnerability detection, compliance), preferred interfaces (e.g., CLI, web-based), input types, acceptable runtime performance (**RQ4**), and pricing models. These questions provided insights into practical challenges and preferences.
- **Confidence in Security Analyzer Outputs (Q16–Q20)**: Focused on factors influencing trust in analyzer results (**RQ5**), this section assessed respondents' confidence levels, reasons for trusting or distrusting tools, and instances where vulnerabilities were ignored (**RQ6**). A free-response question (Q20) allowed participants to describe specific scenarios, offering qualitative context.
- **Impact of Explanation on Confidence (Q21–Q25)**: This section investigated the role of explanation quality in building trust (**RQ7**), asking about preferred explanation formats, the impact of explanations on confidence, and suggestions for improving analyzer outputs. Responses highlighted the types of explanations developers find most actionable.

4.2 Survey Results

To analyze the quantitative survey data, we have employed descriptive statistics. For qualitative insights, two authors conducted a Modified-Delphi card sorting [37] to systematically categorize open-ended responses.

4.2.1 Participant Background. Figure 4 shows that most respondents are smart contract developers (44.67%) and testers (31.33%). A significant majority (75.33%) reported 1–5 years of experience, with 38% having 1–2 years and 37.33% having 3–5 years of experience. Their primary areas of work are NFT projects (61.33%), DeFi platforms (52.67%), and decentralized applications (DApps)/Web3 platforms (44%). Most participants (96%) perceive smart contract security as important. Therefore, more than half (54.66%) have used security analyzers for more than 75% of their work, primarily during audits (78%), coding (75.33%), and testing phases (74%). The primary reasons for using security analyzers include identifying vulnerabilities (89.33%), ensuring code quality (82%), learning about potential security issues (68%), and complying with organizational or regulatory policies (53.33%). Preferences for tool interfaces included command-line interfaces (34%), Integrated Development Environment (IDE) plugins (28%), and web-based tools (21.33%), with a notable inclination toward freemium pricing models (52%).

*4.2.2 Most Common Security Vulnerabilities (**RQ3**).* Figure 5 shows that reentrancy attacks (62.67%) and front-running (62%) are the most familiar vulnerabilities to participants, followed by denial of service attacks (56%), logic errors (55.33%), parity wallet hack 1

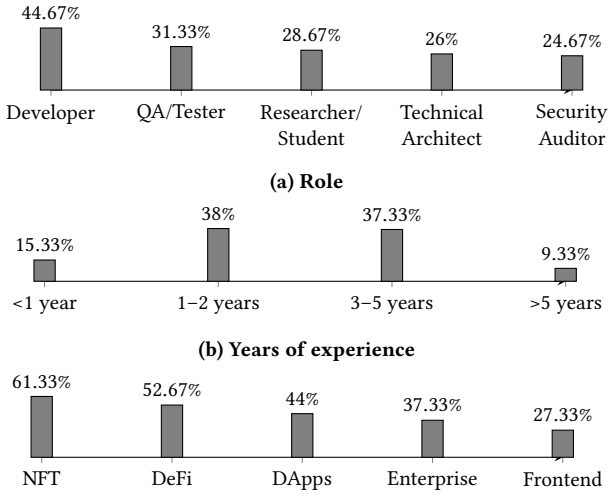


Figure 4: Background of survey participants.

(55.33%), integer overflow/underflow (54.67%), and unhandled exceptions (51.33%). Less familiar vulnerabilities include parity wallet hack 2 (37.33%), flash loan attacks (36.67%), oracle manipulation (28.67%), and greedy contracts (24%). These findings indicate a knowledge gap in emerging threats, which underscores the prevalence of well-documented issues and support the objective of detecting both established and lesser-known vulnerabilities.

> **RQ3**: While participants are familiar with established smart contract vulnerabilities (e.g., reentrancy), they lack enough knowledge about emerging and less common threats (e.g., flash loan attacks).

*4.2.3 Acceptable Runtime Performance of a Security Analyzer (**RQ4**).* Figure 6 shows that the majority of respondents (68%) expect results within 10 minutes: 34% prefer 1–5 minutes, 31.33% favor 5–10 minutes, and 2.67% desire under 1 minute. Additionally, 32% find 10–30 minutes acceptable, but none tolerate runtimes exceeding 30 minutes. These results underscore the demand for efficient tools that deliver quick feedback, because long analysis times disrupt development workflows and diminish tool utility.

> **RQ4**: More than two thirds of participants expect analysis results within 10 minutes.

*4.2.4 Factors Affecting Developer Confidence in Analyzers (**RQ5**).* The majority of developers (73.33%) expressed confidence in analyzer outputs but verified critical findings manually, with only 11.33% fully trusting analysis results without verification. Table 5 shows that providing detailed explanations enhances confidence, as evidenced by an average increase of 1.89 points on a 0–4 scale (0 = "not confident at all," 4 = "very confident"). This shift indicates that, on average, developers transition from "somewhat not confident" to "somewhat confident" when the analyzer provides explanations to reason about its findings. For instance, with explanations, 56% of

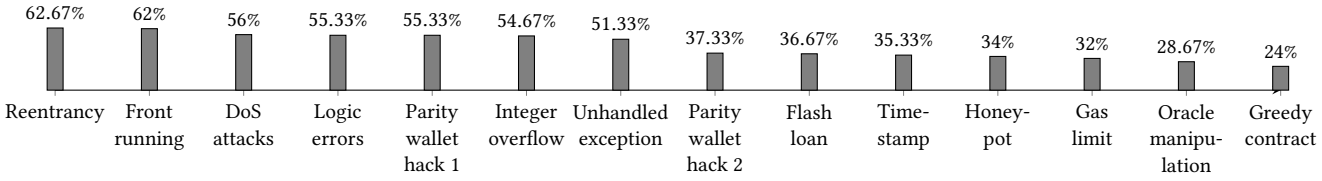


Figure 5: Familiarity with smart contract security vulnerabilities.

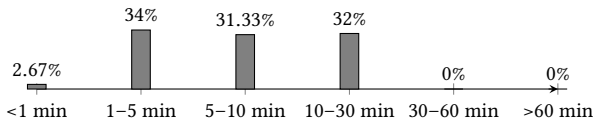


Figure 6: Acceptable running time for security analyzers.

Table 5: Effect of explanations on confidence in security analyzer results.

Confidence Level	Scale	With Explanations	Without Explanations
Not confident at all	0	0.00%	8.00%
Somewhat not confident	1	0.67%	46.67%
Neutral	2	0.67%	18.00%
Somewhat confident	3	42.67%	27.33%
Very confident	4	56.00%	0.00%

participants feel "very confident" compared to only 27.33% feeling "somewhat confident" and 46.67% "somewhat not confident" when no explanations are provided. Overall, 87.33% have higher confidence with explanations, with 50% experiencing an increase of at least 2 points.

Figure 7 presents the main factors that enhance confidence: detailed explanations (82.67%) and analyzer reputation (82%). Confidence is undermined by outdated tools (71.33%), high FPRs (70.67%), and lack of explanations (65.33%). The quality of explanations influences confidence for 58.67% of developers. These findings highlight the necessity of clear and reliable reporting of flagged vulnerabilities.

> **RQ5**: More than half of our participants are more confident when analyzers provide explanations for their results. On the other hand, outdated tools, high FPRs, and lack of explanations undermine user trust.

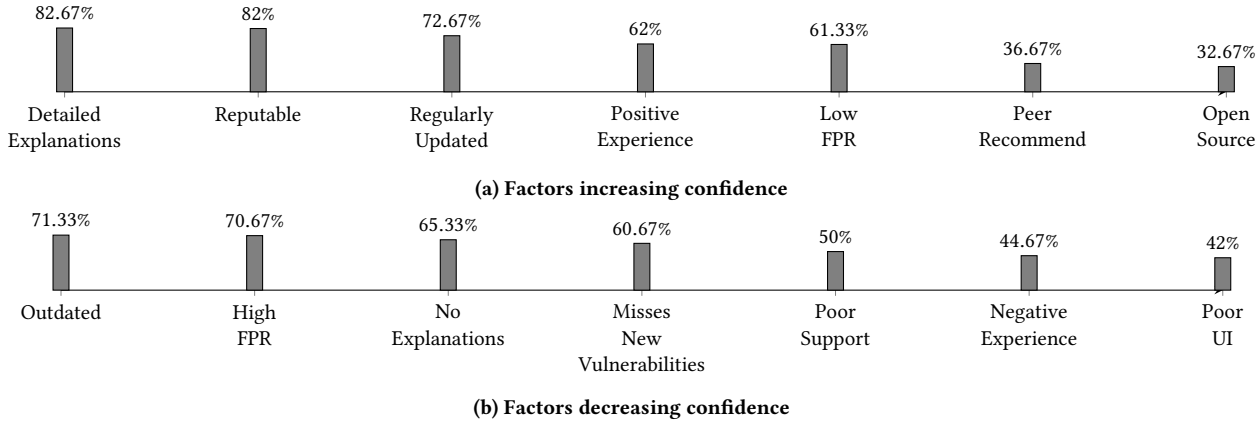
*4.2.5 Reasons Developers Ignore Reported Vulnerabilities (**RQ6**).* Our survey investigated why developers disregard vulnerabilities flagged by smart contract security analyzers, addressing this through closed-ended (Q19) and open-ended (Q20) questions. In Q19, 34.67% of 150 respondents reported never ignoring warnings, while 33.33% cited excessive false positives, 18.67% pointed to unclear or inadequate explanations, and 12% noted time constraints as reasons for dismissal. To gain deeper insights, we conducted a qualitative analysis of 150 open-ended responses from Q20, coding prevalent themes to identify common rationales. The most frequent themes were "manual verification" (14.67%, reflecting the effort needed to validate warnings), "false positives" (14%, indicating distrust in tool accuracy), "low risk or severity" (10%), and "analyzer limitations" (8%, e.g., lack of context in reports). Less common themes included "trust in alternative security measures" (2%), "time constraints" (1.33%), and "focus on critical issues" (1.33%). One response (0.67%) was deemed irrelevant to smart contract analyzers, and 10% provided no actionable rationale (coded as N/A). Among responses, 22.67% expressed mixed rationales, most commonly combining "false positives" (29.17% of mixed responses) and "manual verification" (27.78%), followed by "low risk" and "time constraints" (13.89% each). Additionally, 15.33% of Q20 respondents reiterated never ignoring warnings, a subset of the 34.67% from Q19, as Q20's open-ended format allowed some to reaffirm this stance while others described hypothetical scenarios (e.g., high false positives or unclear reports) that might lead them to ignore warnings. This distinction highlights that while 34.67% consistently avoid ignoring warnings, a smaller group (15.33%) explicitly reaffirmed this in Q20, with others noting potential reasons for future dismissal. The inter-rater reliability for coding Q20 responses was high (Cohen's $\kappa = 0.866$), exceeding the 0.81 threshold for almost perfect agreement [28], ensuring the robustness of our qualitative analysis.

> **RQ6**: Excessive false positives and unclear explanations are primary reasons why developers often disregard vulnerabilities reported by security analyzers.

*4.2.6 Types of Explanations Developers Find Most Useful (**RQ7**).* To understand developers' preferences for vulnerability explanation formats, we analyzed responses to survey questions Q21 and Q25. Quantitatively, our analysis of Q21 shows that a significant majority of respondents (82%) favors brief vulnerability descriptions and fix suggestions, closely followed by a preference for precise bug locations (78%).

To identify specific preferences, we have conducted a qualitative analysis of on 150 open-ended responses from Q25. Standalone preferences, appearing as the sole desired feature in a response, include: "structured explanations" (8%), "contextualized explanations" (6.67%), "actionable remediation steps" (6%), and "visual aids" (3.33%). Other features, such as "integration with development tools", "prioritized severity", "real-world examples and documentation", and "transparency or analyzer accuracy", are mentioned by ≤1.33% of respondents each. We coded 12.67% of the responses as N/A.

The majority of responses (58.67%) exhibit a "mixed rationale", indicating a desire for a combination of explanation features. Within these mixed responses, the most frequently identified preferences



(a) Factors increasing confidence

(b) Factors decreasing confidence

Figure 7: Factors affecting confidence in the results of security analyzer.

are "actionable remediation steps" (20.15%) and "contextualized explanations" (17.16%), followed by "real-world examples and documentation" (17.16%), "visual aid" (13.81%), and "structured explanations" (11.94%). The inter-rater reliability for coding the responses to Q20 is relatively high (Cohen's $\kappa = 0.755$), which is close to the 0.81 threshold that typically indicates substantial agreement between independent annotators [28].

> **RQ7**: The most preferred features in smart contract analyzers include highly value brief vulnerability descriptions, fix suggestions, and precise bug locations, with actionable remediation steps and contextualized explanations. More than half of our participants indicate a set of desirable features that often combines actionable steps, contextualized explanations, real-world examples, visual aids, and structured formats.

4.3 Implications for Developers of Security Analyzers for Smart Contracts

Based on our survey of 150 participants, we distill the following design recommendations for improving smart contract security analyzers. These actionable principles directly reflect user needs and expectations, supporting more effective, trustworthy, and widely adopted tools.

(1) *Expand Vulnerability Coverage.* Users report high awareness of established vulnerabilities such as reentrancy (62.67%) but lower awareness of emerging threats such as flash loan attacks (36.67%) and oracle manipulation (28.67%). To provide comprehensive coverage and raise developer awareness of evolving risks, security analyzers should incorporate detection mechanisms for both well-known and emerging vulnerabilities.
(2) *Prioritize Rapid Feedback.* More than two thirds of our survey participants expect analysis results within 10 minutes, and none accept runtimes over 30 minutes. Tools should aim to deliver results in seconds rather than minutes, enabling integration into iterative development workflows without disrupting productivity.
(3) *Build Trust Through Detailed Explanations.* More than half of our participants are more confident in tool outputs when detailed explanations are provided. Therefore, analyzers should generate clear, interpretable reports that minimize false positives and provide concrete, verifiable justifications for flagged vulnerabilities.
(4) *Reduce Ignored Warnings by Enhancing Report Quality.* The main reasons users ignore warnings are high false positives (33.33%) and unclear explanations (18.67%). Security tools must focus on precision and clarity, offering context-specific, actionable reports that reduce manual verification overhead and increase developer engagement with reported issues.
(5) *Provide Actionable Fix Suggestions.* More than 80% of participants prefer vulnerability reports that include both brief descriptions and actionable fix suggestions. Therefore, tools should present multifaceted reports combining technical explanations with clear remediation steps, improving both usability and security outcomes.
(6) *Support Multiple Interaction Modes.* Participants reported varied interface preferences: command-line interfaces (34%), IDE plugins (28%), and web-based tools (21.33%). To maximize adoption, analyzers should support flexible deployment options accommodating diverse developer workflows.

By aligning tool design with these survey-driven insights, security analyzers can better meet developer expectations and foster more secure smart contract development practices.

5 Threats to Validity

Following established empirical research frameworks [12, 20], we will discuss potential threats to the validity of our study, categorized into internal, external, and construct validity. For each category, we identify key threats, assess their potential impact, and describe mitigation strategies that we have employed to ensure the robustness of our findings.

5.1 Internal Validity

A primary threat arises from the accuracy of ground truth labels for vulnerabilities in our curated dataset. Inaccuracies in these labels

might bias our valuation metrics such as precision and recall, potentially skewing our assessment of tool performance. To mitigate this, we conducted a manual code analysis to identify vulnerability patterns, reviewed transaction histories to detect evidence of exploitation, and cross-validated labels against reported incidents from security audits and academic literature [1, 11, 22, 25] and ensured that the dataset includes both exploited and unexploited contracts, adhering to the principle: *vulnerability does not imply exploitation* [38].

Another threat is the 900-second timeout set for tool evaluations, which might truncate analysis for slower tools (e.g., Mythril and Confuzzius), affecting performance metrics. However, this limit 50% more than the 10-minute window within which developers expect to receive analysis results. Therefore, we see it is a reasonable balance between completeness and practical constraints.

Finally, self-reported data from our survey may be subject to social desirability bias, potentially inflating reported expertise or tool usage. To address this issue, we ensured anonymity through Prolific [39] and targeted experienced practitioners to encourage candid responses.

5.2 External Validity

Our dataset of Ethereum contracts may not fully represent contracts on other blockchains (e.g., Binance Smart Chain [4]), potentially limiting the applicability of our results. To address this threat, we ensured that our dataset included diverse vulnerability types and contract states that reflect real-world scenarios.

Similarly, our survey's focus on Ethereum developers may restrict broader applicability to other blockchain ecosystems. To mitigate this threat, we recruited a diverse sample of practitioners with different roles and experience levels, representing a broad cross-section of the Ethereum community.

Furthermore, our findings for the five tools evaluated may not extend to other tools or future versions. To counter this threat, we selected widely-used, actively maintained tools employing diverse analysis techniques.

5.3 Construct Validity

A key threat is that our definitions of vulnerabilities may differ across practitioner interpretations, potentially misaligning our metrics with real-world perceptions. To mitigate this threat, we adopted standardized definitions from established literature [3, 48]. We also focused on well-documented issues validated by security audits.

Another concern is that developer trust, measured through self-reported confidence and qualitative responses, may not fully reflect actual behavior. To address this issue, we employed a mixed-methods approach, combining quantitative scales with qualitative open-ended questions. Our approach achieves high inter-rater reliability for qualitative coding (Cohen's $\kappa = 0.866$ for Q20, $\kappa = 0.755$ for Q25).

While metrics such as precision, recall, and F1-score evaluate technical performance, they may not fully capture usability or real-world effectiveness. To address this issue, we incorporated multiple metrics, including false negative rate, false positive rate, and analysis time. We then complemented these metrics with survey insights on trust and usability to provide a comprehensive assessment.

Systematically addressing these threats through rigorous design, validation, and mitigation strategies, we enhance the reliability and credibility of our study's conclusions. While no study is without limitations, these measures minimize biases and strengthen our contributions to smart contract security research.

6 Related Work

6.1 Smart Contract Vulnerabilities

Research on smart contract vulnerabilities has established a foundation for understanding their nature and impact. Atzei et al. [3] provided a seminal taxonomy of common vulnerabilities, including reentrancy, integer overflow/underflow, and unchecked external calls, which continue to threaten contract integrity. Zhou et al. [48] built on this work by applying it to smart contract security, identifying over 20 vulnerability types and emphasizing the need for automated detection mechanisms. Real-world incidents (e.g., the DAO hack in 2016 [43] and the suicide attack in 2017 that froze US$150 million [42]) illustrate the severe consequences of these flaws. Our study leverages a dataset of 653 real-world contracts exhibiting reentrancy, suicide, and integer overflow/underflow vulnerabilities, reflecting these well-documented issues to comprehensively evaluate security analyzers.

6.2 Evaluation of Security Analysis Tools

Automated security analyzers are vital for detecting vulnerabilities in smart contracts, and their performance has been a focal point of prior research. Durieux et al. [17] conducted an empirical evaluation of nine tools across 69 curated and 47,518 verified contracts, measuring precision and recall. However, that study did not assess tool effectiveness against real-world exploits or incorporate developer feedback. Perez and Livshits [38] examined the gap between flagged vulnerabilities and actual exploits, analyzing 6 academic datasets against Ethereum transaction traces, revealing that only 1.98% of flagged contracts were exploited. While their work is insightful, it did not evaluate the tools themselves or explore developer trust.

Ghaleb and Pattabiraman [21] proposed SolidiFI, a mutation testing framework to evaluate six static analyzers on 50 smart contracts injected with 9,369 distinct vulnerabilities. Their results show that only Slither detected all injected vulnerabilities, while other tools missed several instances despite claimed capabilities. The authors also reported high false positive rates, though manual verification was limited to vulnerabilities missed by most tools, potentially underestimating false positives. Unlike our work, that study examined a smaller, synthetic dataset and did not explore developer perspectives.

Chaliasos et al. [10] evaluated 5 prominent tools on 127 documented DeFi attacks, totaling over US$2.3 billion in losses, and found that these tools could have prevented only 8% of attacks, primarily reentrancy-based. The authors complemented this evaluation with a survey of 49 auditors and developers on tool usage and workflows. However, they did not delve into how tool performance affects trust in reported vulnerabilities.

Our study evaluates five widely-used tools on 653 real-world contracts, assessing precision, recall, false positive rates (FPR), false negative rates (FNR), and efficiency across reentrancy, suicide, and integer overflow/underflow vulnerabilities. Unlike Chaliasos et al.

[10], which focused primarily on detection rates, our measurements reveal high FPRs (e.g., 17.8% for Slither on reentrancy, 26.7% for Oyente) that impose significant verification burdens on developers, as confirmed by our survey of 150 practitioners, where 33.33% cited false positives as a reason for ignoring warnings. Additionally, our inclusion of suicide vulnerabilities and a larger, more diverse dataset provides a broader analysis of tool limitations, while our survey offers deeper insights into trust factors, revealing that high false alarm rates and unclear explanations erode confidence. This comprehensive approach, combining extensive benchmarking with developer perspectives, distinguishes our work by addressing both technical and human-centric challenges in smart contract security.

6.3 Developer Trust and Usability

While technical evaluations are essential, developer trust and usability significantly influence the practical adoption of security tools. Chaliasos et al. [10] initiated exploration into developer experiences, surveying 49 practitioners on general tool usage. However, their study does not emphasize trust-specific factors. Similarly, Ivanov et al. [26] catalog tools and techniques but seldom address human-centric challenges. In contrast, Noller et al. [33] conducted a survey with more than 100 software practitioners to investigate factors enhancing developer trust in automatically generated patches for general programming languages, focusing on preferences for interaction, evidence, and repair tool outputs. However, no comparable study exists for smart contracts, a gap that our work fills by understanding developer trust in security tools tailored to this domain.

7 Conclusion

This study presents a comprehensive analysis of smart contract security tools, combining an empirical evaluation of five prominent analyzers with a developer-centric survey. Our results show that no single tool consistently excels across all vulnerability types. For example, Confuzzius performs best for reentrancy detection and Slither achieves the lowest false negative rates. However, they struggle with suicide attacks and show mixed performance for integer arithmetic issues. Mythril's high false negative rate and long analysis times further highlight critical limitations in scalability. Complementing these findings, our survey of 150 smart contract developers reveals that the main usability challenges include high false positives, unclear explanations, and limited support for emerging vulnerabilities. These issues are the primary barriers to developer trust and tool adoption. Developers emphasize the need for rapid feedback, actionable explanations, and flexible interfaces, highlighting gaps between current tool capabilities and real-world development needs.

Our findings point to clear priorities for the smart contract security community. Tool developers should focus on integrating hybrid analysis techniques, improving explanation quality, reducing false positives, and optimizing analysis times to meet developer expectations. Given the variability in tool performance, smart contract developers are encouraged to combine multiple analyzers to ensure broader vulnerability coverage. At the research level, there is significant opportunity to explore ensemble approaches, machine learning integration, and detection of emerging vulnerabilities (e.g., flash loan attacks and oracle manipulation). To accelerate further progress, we call on the community to collaborate on shared benchmarks and open datasets. By addressing both technical limitations and human factors, we hope that security analyzers would become more reliable, usable, and trusted, ultimately strengthening the security of blockchain ecosystems.

References

[1] Tamer Abdelaziz and Aquinas Hobor. 2023. Smart learning to find dumb contracts. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1775–1792.

[2] Anonymous. 2025. Replication Package for "Empirical Evaluation of Security Analyzers" and "Surveying Smart Contract Developers". https://github.com/blockchain-security-artifacts/sc-developer-study-1.

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6*. Springer, 164–186.

[4] Binance. 2020. Binance Smart Chain: A Parallel Blockchain to Binance Chain. https://docs.bnbchain.org/. Accessed: 2025-07-18.

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

[6] Vitalik Buterin et al. 2013. Ethereum white paper. *GitHub repository* 1, 22-23 (2013), 5–7.

[7] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* 3, 37 (2014).

[8] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[9] Chainalysis. 2025. 2025 Crypto Crime Report. https://go.chainalysis.com/2025-Crypto-Crime-Report.html. Accessed: 2025-03-01.

[10] 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.

[11] Jiachi Chen, Xin Xia, David Lo, and John Grundy. 2021. Why do smart contracts self-destruct? investigating the selfdestruct function on ethereum. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2021), 1–37.

[12] Daniela S Cruzes and Lotfi ben Othmane. 2017. Threats to validity in empirical software security research. In *Empirical research for software security*. CRC Press, 275–300.

[13] Leonardo De Moura and Nikolaj Bjørner. 2007. Efficient E-matching for SMT solvers. In *International Conference on Automated Deduction*. Springer, 183–198.

[14] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[15] ConsenSys Diligence. [n. d.]. ConsenSys Diligence - Smart Contract Audits. https://consensys.net/diligence/. Accessed: 2025-07-18.

[16] Lisa Nguyen Quang Do, James R Wright, and Karim Ali. 2020. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering* 48, 3 (2020), 835–847.

[17] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*. 530–541.

[18] Ethereum. 2025. Etherscan: The Ethereum Blockchain Explorer. https://etherscan.io/.

[19] 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.

[20] Robert Feldt and Ana Magazinius. 2010. Validity threats in empirical software engineering research-an initial survey.. In *Seke*. 374–379.

[21] Asem Ghaleb and Karthik Pattabiraman. 2020. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 415–427.

[22] GitHub. 2022. SB Curated. https://github.com/smartbugs/smartbugs-curated Accessed: Jul 16, 2025.

[23] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 557–560.

[24] Jiaqi He, Revan MacQueen, Natalie Bombardieri, Karim Ali, James R. Wright, and Cristina Cifuentes. 2023. Finding an Optimal Set of Static Analyzers To Detect Software Vulnerabilities. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 463–473.

[25] Tianyuan Hu, Jingyue Li, Bixin Li, and Andre Storhaug. 2024. Why smart contracts reported as vulnerable were not exploited? *IEEE Transactions on Dependable and Secure Computing* (2024).

[26] Nikolay Ivanov, Chenning Li, Qiben Yan, Zhiyuan Sun, Zhichao Cao, and Xiapu Luo. 2023. Security threat mitigation for smart contracts: A comprehensive survey. *Comput. Surveys* 55, 14s (2023), 1–37.

[27] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681.

[28] J. R. Landis and G. G. Koch. 1977. The measurement of observer agreement for categorical data. In *Biometrics*. 159–174.

[29] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 254–269.

[30] MarketsandMarkets. 2024. Blockchain Market by Component, Provider, Type, Organization Size, Application, and Region - Global Forecast to 2029. https://www.marketsandmarkets.com/Market-Reports/blockchain-technology-market-90100890.html. Accessed: 2025-03-15.

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

[32] Bernhard Mueller. 2018. Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam* 9 (2018), 54.

[33] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust enhancement issues in program repair. In *Proceedings of the 44th international conference on software engineering*. 2228–2240.

[34] Trail of Bits. [n. d.]. Trail of Bits - Security Audits and Research. https://www.trailofbits.com/. Accessed: 2025-07-18.

[35] OpenZeppelin. [n. d.]. OpenZeppelin - Audits and Open Source Security. https://openzeppelin.com/security-audits/. Accessed: 2025-07-18.

[36] OpenZeppelin. 2025. OpenZeppelin Contracts. https://github.com/OpenZeppelin/openzeppelin-contracts.

[37] Celeste Lyn Paul. 2008. A modified delphi approach to a new card sorting methodology. *Journal of Usability studies* 4, 1 (2008), 7–30.

[38] Daniel Perez and Benjamin Livshits. 2021. Smart contract vulnerabilities: Vulnerable does not imply exploited. In *30th USENIX Security Symposium (USENIX Security 21)*. 1325–1341.

[39] Prolific. 2025. Prolific: Participant Recruitment for Research. https://www.prolific.com/. Accessed: 2025-02-01.

[40] SWC Registry. 2020. Integer Overflow and Underflow. https://swcregistry.io/docs/SWC-101/ Accessed: Jul 16, 2025.

[41] SWC Registry. 2020. Reentrancy. https://swcregistry.io/docs/SWC-107/ Accessed: Jul 16, 2025.

[42] SWC Registry. 2020. Suicide Attack. https://swcregistry.io/docs/SWC-106/ Accessed: Jul 16, 2025.

[43] David Siegel. 2016. Understanding the DAO attack. https://www.coindesk.com/learn/understanding-the-dao-attack.

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

[45] Christof Ferreira Torres, Julian Sch¨utte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th annual computer security applications conference*. 664–676.

[46] Qin Wang, Rujia Li, Qi Wang, and Shiping Chen. 2021. Non-fungible token (NFT): Overview, evaluation, opportunities and challenges. *arXiv preprint arXiv:2105.07447* (2021).

[47] Sam Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William Knottenbelt. 2022. Sok: Decentralized finance (defi). In *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*. 30–46.

[48] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. 2023. Sok: Decentralized finance (defi) attacks. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2444–2461.