Smart Contract and DeFi Security: Insights from Tool Evaluations and Practitioner Surveys

Stefanos Chaliasos
Imperial College London
United Kingdom

Marcos Antonios Charalambous
Imperial College London
United Kingdom

Liyi Zhou
Imperial College London
United Kingdom

Rafaila Galanopoulou
National and Kapodistrian University of Athens
Greece

Arthur Gervais
University College London
United Kingdom

Dimitris Mitropoulos
National and Kapodistrian University of Athens
Greece

Benjamin Livshits
Imperial College London
United Kingdom

ABSTRACT

The growth of the decentralized finance (DeFi) ecosystem built on blockchain technology and smart contracts has led to an increased demand for secure and reliable smart contract development. However, attacks targeting smart contracts are increasing, causing an estimated $6.45 billion in financial losses. Researchers have proposed various automated security tools to detect vulnerabilities, but their real-world impact remains uncertain.

In this paper, we aim to shed light on the effectiveness of automated security tools in identifying vulnerabilities that can lead to high-profile attacks, and their overall usage within the industry. Our comprehensive study encompasses an evaluation of five SoTA automated security tools, an analysis of 127 high-impact real-world attacks resulting in $2.3 billion in losses, and a survey of 49 developers and auditors working in leading DeFi protocols. Our findings reveal a stark reality: the tools could have prevented a mere 8% of the attacks in our dataset, amounting to $149 million out of the $2.3 billion in losses. Notably, all preventable attacks were related to reentrancy vulnerabilities. Furthermore, practitioners distinguish logic-related bugs and protocol layer vulnerabilities as significant threats that are not adequately addressed by existing security tools. Our results emphasize the need to develop specialized tools catering to the distinct demands and expectations of developers and auditors. Further, our study highlights the necessity for continuous advancements in security tools to effectively tackle the ever-evolving challenges confronting the DeFi ecosystem.

1 INTRODUCTION

The emergence of Ethereum and blockchains with smart contract capabilities led to the development of decentralized applications (dapps), opening up new possibilities for innovation. The Decentralized Finance (DeFi) ecosystem, which is built on these technologies, has experienced significant growth since 2020, with the total value locked (TVL) reaching an all-time high of 180 billion USD on December 2021 [2]. Unfortunately, this massive amount of value locked in DeFi has also made them an attractive attack target. Despite the efforts to write secure dapps, attackers have successfully exploited vulnerable smart contracts causing losses of 6.45 billion dollars [2], underscoring the need for effective security measures.

Over the years, researchers have dedicated tremendous efforts to secure smart contracts by developing new techniques and tools that identify vulnerabilities [22, 27]. Such techniques involve static analysis [5, 6, 16, 45], symbolic execution [11, 30], fuzzing [19, 23, 44, 51], formal verification [35, 43], runtime verification [40], and machine learning-based [29, 53]. Despite these efforts, high-profile attacks on smart contracts still persist. To understand and evaluate these approaches, researchers have conducted various studies. Durieux et al. [14] and Ren et al. [39] evaluated smart contract security tools, while Perez and Livshits [34] assessed the high number of false positives of automated security tools. Additionally, Zhang et al. [57] performed a systematic investigation to highlight missing vulnerability oracles.

Although there has been significant research and focus on smart contract security, it remains unclear how effective automated security tools are against real-world exploits, what impact these tools have on the industry, and how they are utilized in developing and auditing smart contracts. In this paper, we aim to answer the following research questions.

RQ1: Which vulnerability types can be detected by automated security tools? How frequently do these vulnerabilities occur in real-world attacks? What is the severity level of the vulnerabilities that could have been detected by automated security tools in real-world attacks? Finally, what types of vulnerabilities cannot be detected by current automated security tools? (Section 4.1)

RQ2: To what extent can security tools be used to prevent real-world high-profile attacks? What is the effectiveness of automated security tools against each vulnerability category? Which high-profile attacks could have been potentially avoided by using semi-automated security tools that require user input? (Section 4.1)

RQ3: What is the landscape of security tools used by developers and auditors? To what extent do developers prefer open-source tools? How prevalent are academic tools in practice? What percentage of practitioners use semi-automated tools that can prevent specific vulnerability types that are out-of-scope for automated security tools? How much time do auditors typically spend using security tools during audits? (Section 4.2)

RQ4: What are the key characteristics of security tools that are prioritized by auditors and developers? Do practitioners weigh the trade-off between false positives and false negatives when selecting security tools, and how? Additionally, are ease of use, documentation, and report quality important factors when selecting security tools for both developers and auditors? (Section 4.3)

RQ5: How effectively do security tools address various classes of errors according to auditors and developers? Specifically, which types of errors are inadequately covered by current security tools? Additionally, what is the perception of auditors regarding the usefulness of security tools? (Section 4.3)

Methodology. To address RQ1-RQ2, we conducted an extensive empirical evaluation of five automated security tools using a dataset of 127 high-impact real-world attacks. In Section 3.1, we describe the dataset, the selection criteria we followed for the tools, and our benchmarking process. To answer RQ3-RQ5, we conducted surveys with 49 developers and auditors working in top DeFi protocols. Our methodology for performing the surveys is presented in Section 3.2.

Findings. Through our extensive analysis, we have obtained the following findings regarding the current state of security tools' effectiveness and usage in the industry.

RQ1: Our empirical analysis revealed that the selected automated security tools can identify 14 different types of vulnerabilities. Among the attacks in our dataset, a total of 32 out of 127 exploits were associated with vulnerabilities in these 14 categories. These 32 vulnerabilities resulted in a total damage of approximately 271.5 million USD. Notably, the top two types of vulnerabilities in the attack dataset involve concepts such as coding logic or sanity checks or on-chain oracle manipulation, which in turn cannot be detected by current automated security tools.

RQ2: The evaluation indicates that automated security tools could have potentially prevented 11 out of 32 in-scope attacks, resulting in a total loss of 149 million USD. However, security tools tend to generate numerous insignificant reports, leading to a potentially overwhelming number of false positives. All detected vulnerabilities were related to reentrancy, highlighting the effectiveness of security tools against this type of vulnerability but also the inefficiency against other types. Furthermore, our analysis indicates that existing security tools neglect protocol layer vulnerabilities. Interestingly, semi-automated tools could potentially prevent 47 attacks involving code logic absence, sanity checks, and logic errors.

RQ3: Our survey results show that developers tend to use lightweight tools that can be easily integrated into the development life cycle, such as linters, while auditors use more sophisticated tools with greater bug-finding capabilities (e.g., static analyzers). The majority of developers (92%) prefer open-source tools, while over half of the participants reported using in-house security tools. We found that academic tools used in research evaluations and benchmark studies are not commonly used in practice. Furthermore, about 59% of developers and 48% of auditors use tools that can detect logic-related bugs, which are often the root cause of high-impact attacks. The majority of auditors (76%) reported using security tools for up to 20% of their audit time.

RQ4: The results of our survey indicate that developers prefer security tools with low false negative rates, while auditors prefer tools with low false positive rates since they are responsible for triaging reports. In addition, auditors place a greater emphasis on the tool's setup process and its bug-finding capabilities, while developers prioritize tools that can be easily integrated into their development workflows. Both auditors and developers consider ease of use, documentation, and report quality to be important factors when selecting security tools.

RQ5: Our findings reveal that both developers and auditors consider logic-related bugs and oracle manipulation vulnerabilities as significant threats that are inadequately addressed by security tools. They express the need for better support for these types of vulnerabilities. While the over half (52.4%) of auditors find security tools helpful for auditing, a notable proportion (38.1%) do not find them useful, highlighting the need for further improvement in the development and use of security tools for auditing purposes.

2 BACKGROUND

The literature on the evaluation of automated security tools for smart contracts has been primarily focused on assessing their effectiveness by constructing various benchmarks (see Figure 1). Durieux et al. [14] developed Smartbugs, an extendable evaluation framework that facilitates the integration and comparison between multiple security tools that analyze EVM smart contracts. In [14], the authors employed 9 automated analysis tools using two datasets; one consisting of 47K contracts for consistency evaluation, and the other one, 69 annotated vulnerable contracts for precision evaluation. Ren et al. [39] proposed a comprehensive 4-step evaluation process for minimizing bias in the assessment of automated tools.

Contrary to previous work, our study aims to evaluate the real-world impact of automated security tools. Perez and Livshits [34] surveyed 23K smart contracts reported as vulnerable in 6 academic papers and found that only 1.98% of them had been exploited since deployment, highlighting a potentially high number of false positives in existing techniques. In contrast, we focus on assessing automated tools' false negatives and gaining a deeper understanding of their limitations.

Zhang et al. [57] performed a systematic investigation of 462 defects reported in CodeArena audits and 54 exploits to study the extent to which existing tools could detect them. Our work takes a different approach by actually running the tools against exploits and reporting both cases where the tools have false negatives and cases where the tools lacked appropriate oracles. Wan et al. [46] surveyed 156 practitioners to understand their perceptions and practices on smart contract security. Our study on the other hand, focuses on surveying dapp developers and auditors to investigate how they use smart contract security tools.

In contrast to previous studies, this paper presents a mixed-methods investigation into the effectiveness and usage of security tools. The aim is to provide a comprehensive overview of the current status and offer valuable insights for researchers and practitioners to advance the state-of-the-art in smart contract and DeFi security.

Papers	Venue	Dataset	Vulnerabilities	Tools	Running Tools	Exploited Contracts	Measured Impact	Surveyed Practitioners
Durieux et al. [14]	ICSE'20	69 manually annotated vulnerable contracts and 49K on-chain contracts	9	9	✓	✗	✗	✗
Ren et al. [39]	ISSTA'21	176 contracts retrieved from Github, EIPs, Academic papers	6	6	✓	✗	✗	✗
Perez et al. [34]	USENIX SEC'21	23K vulnerable contracts reported by six recent academic projects	6	6	✓	463	1.7M USD	✗
Zhang et al. [57]	ICSE'23	516 from codearena and on-chain exploits	17	38	✗*	54	✗	✗
This work		127 on-chain exploits	39	5	✓	127	2.3B USD	✓

Figure 1: Point-to-point comparison of related work on evaluating automated security tools. *: This paper categorizes bugs in machine unauditable bugs. Two tools, Slither and Oyente, were used not to measure false positives and false negatives, but to validate whether the tools were able to detect MUB bugs as defined in the study..

3 METHODOLOGY

We provide an overview of the methods we employed to evaluate the capability of current security tools to find real-world vulnerabilities and understand practitioners' experience when using such tools. Specifically, we describe the dataset containing real-world exploits, the tool selection criteria, and the benchmarking process. Further, we focus on the design of the surveys, participant demographics, and how we analyzed the results.

3.1 Empirical Evaluation

Dataset. We use the dataset of DeFi attacks presented by Zhou et al. [61] as a basis for our analysis. The dataset includes a comprehensive analysis and classification of 181 real-world, high-impact DeFi attacks. Attack details involve underlying vulnerabilities in smart contracts, corresponding exploits, and monetary losses. The vulnerabilities are categorized into five layers including *Network*, *Consensus*, *Smart Contract*, *DeFi Protocol*, and *Auxiliary Service*. Our work focuses on the Smart Contract and the DeFi Protocol layers, because these are typically the layers where developer errors occur and security tools focus their analyses. Hence, we filtered out all vulnerabilities related to other layers. This resulted in a dataset of 127 attacks. Figure 2 presents the vulnerability types as reported in [61] while Figure 3 depicts the total impact of the corresponding attacks. Additionally, we downloaded the source code [1] and bytecode of the smart contracts that were attack targets.

We chose this dataset because it reflects the real-world attacks that have occurred in the smart contract and DeFi ecosystem. While other related works [14, 39] have employed datasets of known vulnerable contracts or contracts with induced vulnerabilities, we believe that our selection of real-world attacks provides a more representative sample of the types of vulnerabilities smart contract developers and auditors should be aware of. Furthermore, the contracts in the dataset have greater complexity than minimal examples, making reasoning about them more challenging.

Tools Selection. To select the security tools for our study, we first conducted an advanced keyword search on Google Scholar [2] and followed references to identify additional tools. We also searched for security tools in GitHub repositories. The above process resulted in 75 tools.

Vulnerability	Layer	#	Tools				
			Solhint [36]	Slither [16]	Mythril [11]	ConFuzzius [44]	Oyente [30]
Absence of coding logic or sanity check	SC	42					
On-chain oracle manipulation	PRO	29					
Reentrancy	SC	13	●	●	●	●	●
Liquidity borrow, purchase, mint, deposit	PRO	10					
Camouflage a token contract	PRO	9					
Token standard incompatibility	PRO	8					
Function/State Visibility Error	SC	8	●				
Other unsafe DeFi protocol dependency	PRO	7					
Other Inconsistent, improper or unprotected access control	SC	5	●	●	●	●	●
Logic Errors	SC	5					
Unfair slippage protection	PRO	4					
Unfair liquidity providing	PRO	4					
Direct call to untrusted contract	SC	4			●		
Other protocol vulnerabilities	PRO	3					
Governance attack	PRO	3					
Transaction Order Dependence	PRO	2			●	●	●
Other coding mistakes	SC	2					
Delegatecall to Untrusted Callee	SC	2		●	●	●	
Unsafe call to phantom function	PRO	1					
Improper asset locks or frozen asset	SC	1		●		●	
Other unfair or unsafe DeFi protocol interaction	PRO	1					
Camouflage a non-token contract	PRO	1					
Weak Randomness	PRO	0	●	●			
Unhandled or mishandled exception	SC	0	●	●	●	●	●
Unbounded or gas costly operation	SC	0			●		
Timestamp Dependence	PRO	0	●	●	●	●	●
Shadowing State Variables	SC	0		●			
Outdated compiler or solidity version	SC	0	●	●			●
Integer Overflow and Underflow	SC	0		●	●	●	●

Figure 2: Summary of vulnerability categories and the number of corresponding exploits in the Zhou et al. dataset [61]. ● indicates tool support for a corresponding vulnerability type. SC: Smart Contract Layer, PRO: Protocol Layer. We exclude vulnerability types that (1) the tools cannot support and (2) do not exist in the dataset.

Next, we applied a number of criteria to narrow down our selection. Specifically, we focused on (1) the availability of source code (51 tools), (2) maintenance (14 tools), (3) ability to run automatically without input (7 tools). Further, we included at least one tool based on the following techniques: linting, static analysis, fuzzing, and symbolic execution. Notably, focusing on tools that are based on different analyses methods is an important dimension of our study.

[1] via etherscan when available

[2] "Smart contract", "Smart contract security", "Ethereum", "ETH", "Ethereum Virtual Machine", "EVM", "EVM bytecode", "Solidity", "Ethereum automated analysis tools", "Blockchain", "Blockchain security", "Ethereum security", "Ethereum vulnerabilities", "DeFi", "Decentralized Finance"

Attacks	127
Damage	2,331,903,028 $
Attacks out of selected tools' scope	95 (75%)
Damage out of selected tools' scope	2,060,349,987 $ (88%)
Attacks in selected tools' scope	32 (25%)
Damage in selected tools' scope	271,553,041 $ (12%)

Figure 3: Overall descriptive statistics of the analysed attacks.

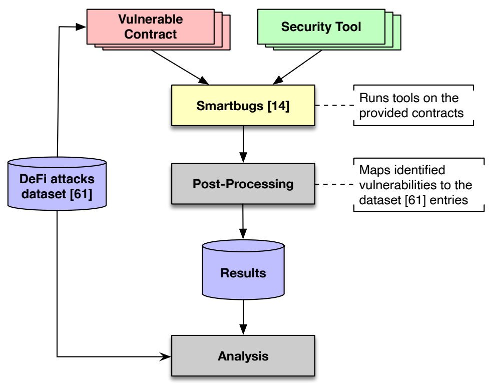

Vulnerable Contract
Security Tool
Smartbugs [14]
Runs tools on the provided contracts
DeFi attacks dataset [61]
Post-Processing
Maps identified vulnerabilities to the dataset [61] entries
Results
Analysis

Figure 4: Evaluating the effectiveness of security tools.

Based on the above criteria, we ended up with the following tools: *ConFuzzius* [44], *Mythril* [11], *Oyente* [30], *Slither* [16], and *Solhint* [36]. Solhint, Slither, and Mythril are widely recognized as the most popular and up-to-date linter, static analyzer, and symbolic executor, respectively. ConFuzzius is the most updated fuzzer that meets our selection criteria. Despite Oyente not being actively maintained, [3] we chose to include it in our analysis due to its status as one of the earliest academic tools, and its continued use in evaluations of numerous academic works [7, 9, 14, 20, 23, 25, 28, 32, 33, 35, 39, 43–45, 47, 54, 55, 58].

Figure 2 depicts the vulnerabilities that each selected tool can identify. Note that the tools cannot detect every programming error related to a vulnerability type. For example, in the case of "*Other Inconsistent, improper or unprotected access control*", Slither can only detect some of the bugs that can lead to this defect type. In the supplementary material, we provide a comprehensive overview of the tool selection process and a detailed mapping between tool vulnerabilities and the vulnerability categories of Zhou et al. [61].

Benchmarking. Figure 4 summarizes our benchmarking approach. To obtain results from the selected tools we utilized the SmartBugs framework [14] (see also Section 2). [4] Next, we manually tracked all vulnerability types that each tool could detect and mapped them to the vulnerabilities of Figure 2, i.e., the vulnerabilities coming from the dataset. We used a post-processing script to integrate this information with the output of the SmartBugs framework and fed the data into an SQLite database for further analysis. Adding support for more tools is straightforward, as it only requires including the tool in SmartBugs and provide a CSV file that describes the mapping of the tool's detected vulnerabilities to our toolchain.

After retrieving all results, we performed various sanity checks to verify that the results were consistent. In the case of a true positive, two authors independently examined if the result is correct. Solhint identified a number of defects of the following type: *"Function/State Visibility Error"*, in five different exploited contracts. However, all cases were false alarms. Finally, we did not try to manually verify the rest of the results (i.e., potential false positives), and we argue that most of the reports should have been either false positives or vulnerabilities that cannot be exploited in practice, as the contracts in question had millions of USD in TVL, and hence attackers would have had high motivation to attack them.

3.2 Surveys

Protocol. To better understand how developers and auditors perceive and use security tools, we conducted a survey campaign. To do so, we followed Kitchenham and Pfleeger's guidelines [26] (also used in similar studies [10, 46]). Further, we employed best practices [41] to boost practitioner participation. Our survey was anonymous and we made all questions optional. In addition, we added an "other" option where possible to increase response rates. Questions were divided into three categories:

(1) **Demographics** to understand the respondents' background.
(2) **Familiarity/usage of security tools during development and auditing** to assess if practitioners are well acquainted with security tools and how they use them.
(3) **Experience with security tools** to understand how satisfied practitioners are and how these tools can be improved.

To fine-tune our campaign, we performed the following steps. Two authors independently designed two slightly distinct surveys, one for developers and one for auditors. Then, they converged on the questions that should be included in the first versions of the surveys. Moving forward, the first round of the surveys took place with a set of $N = 3$ per survey where we asked the respondents to provide feedback. After that first iteration, we adjusted the questions and performed the same with $N = 5$ per survey. We used the feedback and responses to fine-tune the multiple-choice questions.

Respondent selection and demographics. Our aim was to focus our surveys on practitioners with experience working on protocols with high TVL, which in turn, are the main targets of attackers. Instead of focusing on getting as many responses as possible, we focused on obtaining high-quality responses. Although this strategy might bias our results, it was essential to focus on developers of top protocols and auditors who assess such protocols to understand the direct impact of security tools.

To recruit respondents, we first contacted developers from the top 200 DeFi protocols, as reported by Defillama [2]. For auditors, we looked at the auditing companies with the most audit reports for the top 200 protocols and contacted auditors from those companies. We also contacted the top 100 auditors from Code4Arena [1], as independent auditors are also involved in auditing high-profile projects. We received a total of 49 responses: 27 from developers

[3]There is minimal support from the SmartBugs team fixing various errors, so it can still be used.
[4]We modified SmartBugs to ensure that the latest version of the selected tools was always used, all detectors capable of detecting security vulnerabilities were enabled, and a one-hour timeout per tool per contract was employed. When a tool could accept either source code or bytecode as input, we used both to evaluate its effectiveness.

	Developers	Auditors
Experience in smart contract development/auditing		
more than 5	8 (31%)	5 (23%)
3-5	6 (23%)	3 (14%)
1-2	10 (39%)	6 (27%)
less than one	2 (8%)	8 (36%)
Organization's size		
more than 250	0 (0%)	1 (5%)
50-250	5 (19%)	5 (24%)
26-50	3 (12%)	1 (5%)
6-25	15 (58%)	9 (43%)
1-5	3 (12%)	0 (0%)
Independent	n/a	5 (24%)
Main Targeted blockchains		
Ethereum	25 (93%)	22 (100%)
Polygon	15 (56%)	15 (68%)
Avalanche	6 (22%)	10 (46%)
Arbitrum	11 (41%)	9 (41%)
BSC	7 (26%)	8 (36%)
Fantom	7 (26%)	111 (86%)
Solana	2 (7%)	3 (14%)
Other	6 (22%)	5 (23%)
Status of most mature dapp developed		
Mainet	25 (96.2%)	n/a
Development	1 (3.8%)	n/a
Total	27	22

Figure 5: Survey participant demographics.

Tool	Method	Attacks In Scope	Detected	Damage In Scope	Detected Damage
ConFuzzius	Fuzzing	22	1	$ 256,393,948	$ 25,236,849
Mythril	SE	24	1	$ 263,104,948	$ 25,236,849
Oyente	SE	20	0	$ 247,443,948	$ 0
Slither	SA	20	11	$ 213,793,948	$ 149,792,690
Solhint	Linting	25	0	$ 213,292,041	$ 0
Total		32	11	$ 271,553,041	$ 149,792,690

Figure 6: Tool effectiveness and damage that could have been prevented. SE: Symbolic Execution, SA: Static Analysis.

and 22 from auditors.[5] Figure 5, presents an overview of the demographics of our survey participants.

Data analysis. We analyzed the results based on question types. For multiple-choice and Likert-scale questions, we reported respondent percentages per option. For open-ended questions, we followed an inductive approach in which two authors separately performed open card sorting and regularly discussed emerging themes until an agreement was reached. In the rest of this work, we report percentages given the total responses to each question.

4 RESULTS

In this section, we present the findings of our mixed-method investigation aimed at addressing our research questions.

4.1 Effectiveness and Impact of Security Tools on Real-World Exploits

Recently, automated security tools for detecting vulnerabilities in smart contracts have received increased attention. Previous studies [14, 34, 39] have evaluated their effectiveness by measuring recall and precision on datasets containing contracts sourced from blockchains (e.g., Ethereum) or manually crafted vulnerable contracts. Additionally, Zhang et al. [57] surveyed automated tools to determine their ability to detect various vulnerability categories. However, a key question that remains unanswered is the real-world impact of these tools, particularly in preventing significant exploits. To address this question, we conducted a comprehensive analysis of vulnerabilities in DeFi protocols that have led to significant exploits and assessed the effectiveness of automated security tools in preventing these exploits. Additionally, we quantified the potential funds that could have been saved by utilizing these tools.

Automated tools scope. Figure 2 illustrates the scope of the selected security tools. We find that the automated security tools have oracles for the vulnerabilities that lead to the exploit for only 25% of the 127 attacks studied. These attacks cause a total of 271 M USD in monetary losses, amounting to 12% of the total damage incurred by attacks in the dataset (c.f. Figure 3). Notably, the automated security tools do not have oracles for detecting certain critical vulnerabilities, such as *absence of code logic or sanity checks* and *oracle manipulation*. Conversely, the tools tend to focus on vulnerabilities that do not appear to be frequently targeted by adversaries in high-profile attacks, such as *integer overflows and underflows*, as well as *unhandled or mishandled exceptions* (see Figure 2).

Tool effectiveness on real-world vulnerabilities. Out of 32 attacks that automated security tools can reason about the underlying vulnerabilities, only 11 of them could have been detected and potentially prevented if the tools were used (see Figure 6). [6] Figure 7 depicts the results of the tools. Slither detects the most vulnerabilities, but it also reports many false positives (FP). This can be detrimental to the usability of security tools as the number of reports that cannot lead to exploits may overwhelm users. Furthermore, our evaluation indicates that all tools detect vulnerabilities that were not utilized to exploit the assessed contracts, with static analysis and linting tools reporting a greater number of potential false alarms in comparison to other methods.

Detecting different vulnerability types. Notably, all of the 11 aforementioned attacks were caused by reentrancy vulnerabilities, suggesting that the focus on reentrancy by academic researchers [15, 28, 37, 52] has led to the development of effective tools for this category. Despite the effectiveness of these tools in detecting reentrancy vulnerabilities, there are still major issues. Of the five selected tools, only three were able to detect at least one vulnerability that led to a significant exploit. Additionally, 10 of the vulnerabilities could only be detected by Slither.

Automated security tools (see Figure 2) are unable to detect "*Absence of coding logic or Sanity check*" and "*Logic errors*." Thus, it is crucial to determine how many attacks could have been prevented by tools capable of detecting such errors, such as property-based

[5]Our conjecture is that developers and auditors in this space have not been subjected to a lot of priory surveys.
[6]Given that the tools were available in the time of the attack.

Vulnerability	**Slither**		**Oyente**		**ConFuzzius**		**Mythril**		**Solhint**		**Total**		
	D	ODI	D	ODI	D	ODI	D	ODI	D	ODI	TA	D	ODI
Reentrancy	11	69	0	0	1	1	1	18	0	11	13	11	71
Function/State Visibility Error	✗	✗	✗	✗	✗	✗	✗	✗	5	62	8	5	62
Other inconsistent, improper or unprotected access control	0	8	0	0	0	0	0	9	0	10	5	0	19
Direct call to untrusted contract	✗	✗	✗	✗	✗	✗	0	0	✗	✗	4	0	0
Transaction Order Dependence	✗	✗	0	1	0	0	0	0	✗	✗	2	0	1
Delegatecall to Untrusted Callee	0	5	✗	✗	0	0	0	0	✗	✗	2	0	5
Improper asset locks or frozen asset	0	8	✗	✗	0	0	✗	✗	✗	✗	1	0	20
Weak Randomness	0	11	✗	✗	✗	✗	✗	✗	0	0	0	0	11
Unhandled or mishandled exception	0	84	0	0	0	3	0	25	0	47	0	0	93
Unbounded or gas costly operation	✗	✗	✗	✗	✗	✗	0	17	✗	✗	0	0	17
Timestamp Dependence	0	46	0	0	0	2	0	46	0	55	0	0	69
Shadowing State Variables	0	55	✗	✗	✗	✗	✗	✗	✗	✗	0	0	55
Outdated compiler or solidity version	0	69	0	73	✗	✗	✗	✗	0	103	0	0	109
Integer Overflow and Underflow	0	47	0	5	0	7	0	36	✗	✗	0	0	69

Figure 7: Summary of tool results. D (Detected). ODI (Other Detected Issues): other findings including false positives, defects that cannot be exploited (e.g. in protected functions), or exploitable defects not included in the dataset (i.e., not used in the attacks). TA (Total Attacks).

fuzzers, formal verification, and model-checking tools. Notably, such tools could have potentially prevented 37% (47/127) of the exploits in the dataset, amounting to 1,116,118,649 USD in damage. When combining these tools with automated security tools, the total number of (potentially) preventable exploits in the dataset rises to 75, accounting for 59% of the attacks and 1,359,921,690 USD (58%) of the total damage. Our results complement those of Zhang et al. [?], who found that 79.5% of real-world bugs cannot be detected by automated tools alone. However, their research did not consider the effectiveness of semi-automated tools. Zhang et al. [?] also observed that logical errors often have generalized abstract models, indicating that human involvement could be crucial in constructing testing oracles. This finding is consistent with our preliminary findings. We leave it to future work to evaluate the practical effectiveness of semi-automated tools that can detect logic-related bugs and to assess the difficulty of writing specifications/properties for smart contracts that have been exploited.

Potential preventable losses. Our analysis shows that the total funds that could have been saved if selected tools were employed amount to 149,792,690 USD, highlighting the importance of security tools in protecting smart contracts.

Discussion. Despite almost a decade of research and development, automated security tools are still inefficient in detecting vulnerabilities in real-world contracts with high TVL, while reporting many potentially insignificant issues. Hence, further research is needed to improve the effectiveness and usability of these tools to better protect against financial losses due to vulnerabilities in smart contracts, while it is important to add support for more vulnerabilities.

Conclusions for RQ 1, RQ 2

- In a subset of 32 attacks that automated security tools could have detected, only 11 of the exploited vulnerabilities were detected, highlighting a significant missed opportunity to enhance the security of smart contracts.
- All of the detected vulnerabilities were related to *reentrancy*, indicating the effectiveness of the tools in detecting this type of vulnerability but also highlighting the inefficiency of automated tools in detecting other vulnerabilities.
- The top two types of vulnerabilities, absence of coding logic or sanity checks and on-chain oracle manipulation, cannot be detected by current automated security tools. Moreover, we observe that the majority of protocol-layer vulnerabilities are out of the scope of security tools.
- Semi-automated tools may be able to prevent 47 attacks that involve absence of code logic or sanity checks and logic errors.
- The tools generate many insignificant reports, leading to a potentially overwhelming number of false positives.
- The total funds that could have been saved if the tools were employed are 149,792,690 USD, underscoring the importance of security tools in preventing smart contract vulnerabilities.

Call to action: Security tools should focus on detecting vulnerabilities beyond reentrancy to be more effective in securing smart contracts and DeFi applications.

4.2 Familiarity and Usage of Security Tools

In this section, we aim to explore the role of security tools in the smart contract development lifecycle and DeFi audits, specifically focusing on how practitioners utilize these tools. To address this question, we survey both developers and auditors. In the following, we present the results of the surveys and analyze their implications for the development of secure dapps and effective DeFi audits.

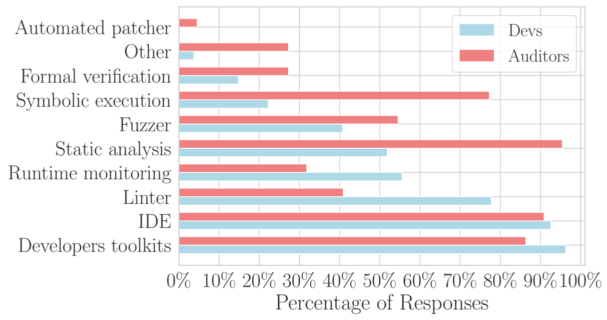


Figure 8: Overall practitioner experience with different tool categories. *Other* includes tools that employ more than one technique.

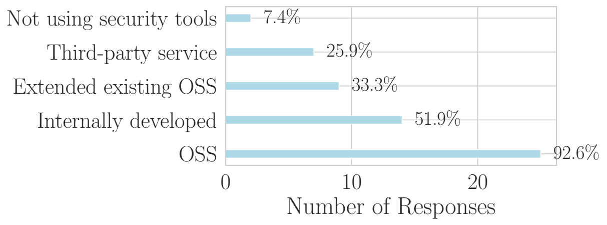


Figure 9: Different tool types used during development.

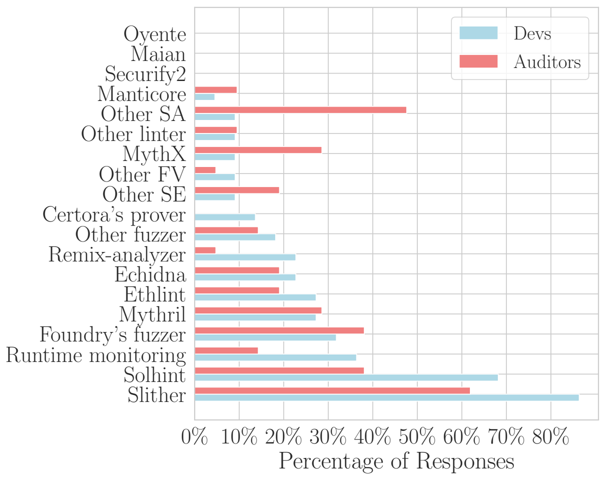


Figure 10: Security tools used by developers and auditors.

Tool familiarity and usage for developers and auditors. Figure 8 illustrates the different types of tools that both developers and auditors have used. The category of tools that most practitioners have used is developer toolkits, followed by IDEs. These tools are primarily used for developing, deploying, debugging, and testing smart contracts. Interestingly, we observe that developers tend to favor lightweight tools such as linters, while auditors prefer tools with greater bug-finding capabilities, such as static analyzers and symbolic executors. Furthermore, developers have more experience using runtime monitoring tools, as most audits are performed on contracts before their deployment. Additionally, we found that developers have used an average of 4.5 different types of tools, while auditors have used an average of 5.3.

Reported tool usage. We further investigated the types of security tools that practitioners use during development to secure decentralized applications (c.f. Figure 9). Only two participants reported that they do not use any tool for this purpose. The majority (92%) of participants report that their organization utilizes open-source tools, and many invest effort into developing internal tools or extending existing open-source tools. Additionally, 25% of participants reported that their organization uses third-party services typically provided by auditing firms. The prevalence of open-source tools highlights the importance of collaboration and community-driven efforts to improve security in the decentralized application ecosystem. Open-source tools have the potential to reach a wider audience and have a greater impact, ultimately leading to more secure and reliable decentralized applications.

Utility of specific tools during development and auditing. Next, we explore which specific tools developers and auditors use during the development and auditing of dapps. Figure 10 displays the results of our investigation. The distribution of tool usage closely mirrors that of Figure 8. It's worth noting that many auditors' responses included "*other*" choices. This is because, in auditing companies, it is common for in-house security analysis tools to be developed and used in audits. Another noteworthy result is that various academic tools, such as Maian, Oyente, and Securify2, which are commonly used in scientific paper evaluations and benchmarking studies [14, 39, 56], are not used in practice. This highlights the need for academia to adapt its comparisons and benchmarks to tools that are actually used by the community.

As we observed in Section 4.1, automated security tools cannot detect logic-related vulnerabilities. Thus, it is crucial to determine how many developers and auditors currently use tools capable of detecting such errors (i.e., formal verification and property-based fuzzing). Our surveys reveal that 59% of developers and 48% of auditors utilize at least one such tool.

Property-based tests and application specifications. As already mentioned, some tools require additional inputs, such as specifications of the smart contracts under test. Hence it is essential to understand who is responsible for providing these inputs. 40% of the respondents indicated developers as responsible for writing specifications/property tests for semi-automated security tools, followed by 29% for auditors and developers, 20% for auditors, and 11% of the respondents were unsure. As the effectiveness of such tools heavily relies on the quality of the provided inputs, we argue that both auditors and developers should participate in that process.

Time spent on tool usage by auditors. Another critical question to consider is how much time auditors spend running, fine-tuning, and validating the results of security tools. The results of our survey, indicate that the majority (76%) of auditors spend a small proportion (between 0%–20%) of their time using such tools. 19% spend between 21% to 40%, while 5% spent 41% to 60%. This suggests that auditing is still primarily a manual effort. While there is certainly potential for tools to improve and automate certain aspects of the auditing process, it will continue to be predominantly a manual effort.

Discussion. As different security tools may use varying techniques, with some being more resource-intensive than others, it is important to match tools appropriately to different stages of the development lifecycle. According to our survey results, developers tend to prefer tooling that can be used during the development process, such as linters, static analyzers, or after deployment, i.e. runtime monitoring tools. Therefore, it is crucial to develop tools that can be easily integrated into developers' daily routines. One such example is Foundry's property-based fuzzer [3], which, despite being a relatively new tool, is already being utilized by a significant number of developers.

Conclusions for RQ 3.

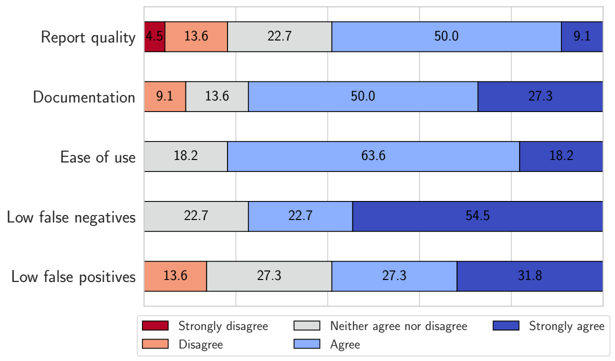
- Overall we observe that developers tend to employ more lightweight tools, including linters, whereas auditors utilize tools with greater bug-finding capabilities (e.g., static analyzers). In addition, developers, use runtime monitoring tools more than auditors.
- Academic tools that appear in the context of research evaluations and benchmark studies such as Oyente, are not used in practice.
- 59% of developers and 48% of auditors utilize tools that can reveal logic-related bugs that are the root cause of many high-impact attacks.
- The majority of auditors (76%) spent only up to 20% of their time using security tools during audits, indicating that the auditing process is mainly a manual effort.

Call to action. To bridge the gap between research and practice, researchers must consider three key factors. First, they should determine if the tools they create will be incorporated into development processes or employed during audits, focusing on prioritizing relevant features. Secondly, emphasizing the detection of vulnerability types that currently cannot be detected by existing security tools is vital. Finally, the evaluation of scientific papers should include benchmarks based on genuine real-world attack scenarios for more accurate and relevant results.

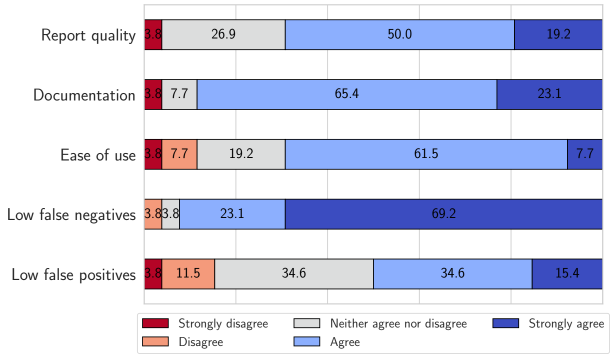
4.3 What Makes Security Tools Valuable to Practitioners

In this section, we aim to understand the factors that practitioners consider important when using security tools. Specifically, we explore the value that security tools provide in the context of detecting smart contract vulnerabilities and assess auditor satisfaction with the results generated by security tools. By examining these aspects, we can gain insight into what makes security tools valuable to practitioners and how they can be further improved to better serve the needs of the DeFi ecosystem.

Importance of tools' characteristics. Results from a Skilert-based question on security tool characteristics are presented in Figure 11. This survey question sought to understand the importance that both auditors and developers place on various aspects of security analysis. The results indicate that both groups consider all of the enumerated characteristics important, but there are some differences in the degree to which each characteristic is prioritized.



(a) Auditors

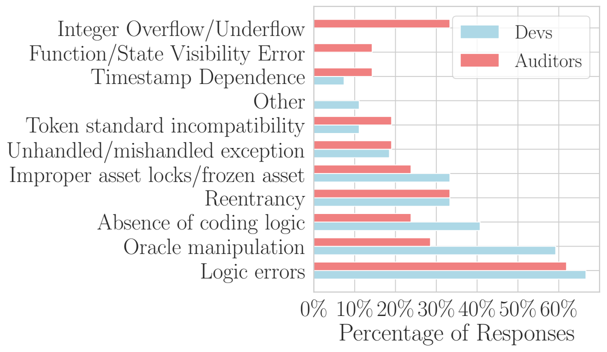


(b) Developers

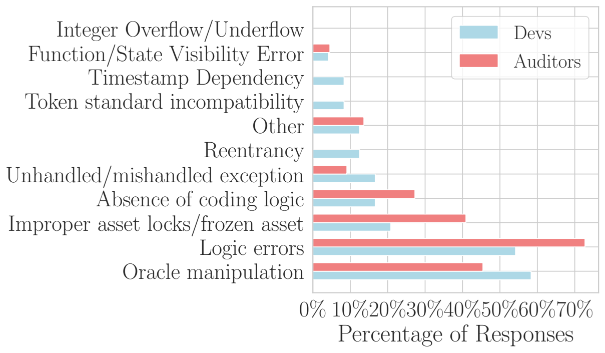
Figure 11: Importance of security tool characteristics.

For developers, low false negatives are perceived as more important mainly because they want reassurance that their applications are safe, whereas, for auditors, low false positives are considered to be more crucial, since it's their job to triage the reports. Additionally, ease of use is a bit more important for auditors, while some auditors do not place as much importance on report quality. Furthermore, we included one open-ended question about other factors that could positively or negatively affect the use of security tools. Many auditors emphasized the importance of tool setup, in addition, to ease of use. One participant highlighted the relation of time to configure / the severity of issues found. For developers, easy integration into the development life cycle (e.g., continuous integration), ease of customization, and the social aspect of other people using the tools and detecting important bugs in real-world applications were the most frequently mentioned factors. Overall, these findings highlight the diverse needs and priorities of practitioners when it comes to security tool features and underscore the importance of developing tools that meet a wide range of requirements.

Exploring practitioners' perspectives on challenging vulnerabilities and available tooling for detecting such vulnerabilities. Figure 12a sheds light on the most challenging vulnerabilities faced by both developers and auditors during the development and manual audit process, respectively. Developers identified logic errors, oracle manipulation, and absence of coding logic as the most difficult vulnerabilities to detect during development, which aligns with the state of most common defects in high-profile real-world attacks (see Figure 2). Auditors identified logic errors as the most



(a) Opinions on the vulnerabilities that are difficult to identify (1) manually (auditors), and (2) during development (developers).



(b) Opinions of developers and auditors on vulnerabilities that are both crucial and cannot be detected by security tools.

Figure 12: Practitioner perspective on security tools and vulnerabilities.

challenging vulnerability to detect manually, followed by several vulnerabilities that existing tools have broad support for, such as integer overflows and reentrancy vulnerabilities, indicating that tools are indeed useful for identifying such bugs.

Regarding vulnerabilities that cannot be detected by automated security tools, both developers and auditors cited oracle manipulation and logic errors as the most challenging (c.f. Figure 12b). Additionally, both groups identified improper asset locks or frozen assets as a vulnerability that requires better support from tools. Overall, these findings emphasize the importance of developing more sophisticated security tools to detect crucial vulnerabilities that current tools may either not support or miss.

How auditors evaluate security tools for auditing smart contracts. Our survey results indicate that a majority of participants found security tools helpful when auditing smart contracts, with 52.4% rating them as 4 or 5 on a 5-point scale. However, a significant portion of respondents (38.1%) did not find security tools to be helpful or found them only somewhat helpful (rated 1-2). This suggests that there is still room for improvement in terms of the effectiveness and usability of security tools. In particular, participants highlighted the need for security tools to address more complex vulnerabilities that pose a greater threat to DeFi applications.

Conclusions for RQ 4 and RQ 5.

- Developers prioritize low false negatives in security tools, while auditors prioritize low false positives in security tools, as it is their job to triage reports. Furthermore, auditors emphasize the importance of tool setup and bug-finding capabilities, while developers emphasize easy integration into the development lifecycle.
- Both developers and auditors want better support for tooling related to logic-related and oracle manipulation vulnerabilities.
- While 52.4% of auditors find security tools helpful for auditing, a significant portion (38.1%) do not find them useful, highlighting the need for further improvement in the development and use of security tools in auditing.

Call to action. Security tools should detect crucial vulnerabilities, such as logic-related and protocol-layer vulnerabilities (e.g., oracle-manipulation bugs), that can result in significant losses in practice. However, it is equally important for security tools to meet high usability standards to be adopted by practitioners.

5 DISCUSSION

5.1 Implications

Effectiveness, coverage, and need for manual inspection. Our analysis shows the limited effectiveness of automated security tools in detecting DeFi vulnerabilities. Figure 2 reveals that only 11 out of 32 (34%) vulnerability types in our dataset were detected by the tools, emphasizing the insufficiency of current automated tools for comprehensive security assurance in DeFi ecosystems. Additionally, given the limited number of vulnerabilities covered by security tools (32/127), smart contract security relies heavily on manual inspections by designers, developers, or auditors. Our survey data indicates that only 59% of developers and 48% of auditors utilize tools capable of identifying logic-related errors, stressing the need for a holistic auditing approach combining automated tools and manual reviews.

Emphasizing semi-automated tools for addressing critical vulnerabilities. Our findings point to the necessity of semi-automated security tools capable of detecting critical vulnerabilities in the smart contract ecosystem. Automated tools, while able to detect reentrancy vulnerabilities, fall short in covering logic-related bugs and protocol-layer vulnerabilities, such as oracle manipulation. Semi-automated tools, which incorporate user input to provide oracles for detecting security issues, present a promising solution. We encourage academia to focus on developing advanced tools that can effectively identify and prevent high-impact vulnerabilities, complementing practical tools already developed by the practitioner community [3, 19].

Fostering collaboration and continuous improvement. Companies should prioritize training and professional development for their developers and auditors. When working with new designs, teams must exercise caution and consider compatibility patterns to minimize vulnerabilities. Collaboration among developers, researchers, and organizations is crucial for staying up-to-date with the latest advancements in smart contract security. This cooperative approach can lead to a better understanding of the current threat landscape and promote the development of more robust security measures.

5.2 Threats to Validity

We use a standard methodology [17] to identify validity threats, which we mitigate where possible.

5.2.1 Empirical analysis.

Internal. One potential threat to internal validity is that the tools' results may be unsound. To mitigate this risk, we cross-checked the results and manually verified the essential findings. We also conducted sanity checks to ensure that the processing of the analysis results was correct. Another potential threat is that the dataset from Zhou et al. [61] may have incorrect data. To address this issue, we manually verified the important findings, while the dataset is open to the public for further verification.

Construct. A potential threat to construct validity is the setup of the tools used in the analysis. To mitigate this risk, we followed the documentation, the setting used in the tool papers, and ran the tools on both source code and bytecode when available. However, we note that ConFuzzius had the highest failure rate per contract, and while we attempted to mitigate any compilation errors, the tool failed in some cases because it could not deploy the targeted contract. ConFuzzius (and fuzzers in general) typically require more fine-tuning per execution, which is out of the scope of this work as we aimed to measure the out-of-shelf solutions available to practitioners with minimal setup. Another potential threat is mapping vulnerabilities from tools to the vulnerability types of Zhou et al. [61]. To address this issue, multiple authors performed the mapping independently, and we iterated over the mapping until reaching an agreement. We further consulted our mapping with the authors of [61].

External. A potential threat to external validity is the size of the DeFi attacks dataset. We used the most extensive dataset with attacks that have fine-grained information. Furthermore, we have automated the whole process, so adding more attacks to the analysis is straightforward. Another potential threat is that we did not include all available tools in the analysis. In this work, we focused on tools most likely to be used by practitioners (c.f. Figure 10). For each analysis technique, we selected the most well-established tool. We further included the most frequently used tool in academic paper evaluations (i.e., Oyente). Finally, we ran several not-maintained academic and industry tools (Securify2, SmartCheck, Conkas, Maian) and observed that their results did not change the paper's overall conclusions.

5.2.2 Surveys.

Internal. Our survey responses may be subject to a potential threat to internal validity, as some respondents may not understand some of the questions well. To reduce this risk, we highlighted in the invitation message that all questions are optional, and that they can skip any question that they do not understand. Additionally, to mitigate this threat, we designed our survey in an iterative fashion, as discussed in Section 3.2).

Construct. Our goal is to survey developers and practitioners that work on projects with high TVL that are typically the targets of adversaries. Hence, we meticulously selected who to invite and did not share our surveys on social media or email lists to focus on the quality of responses rather than quantity.

External. Focusing on developers and auditors who work on top protocols and are more experienced with security tools can pose an external validity threat. This is because our results might not represent the broader ecosystem. Another risk involves the fact that a large percentage of the participants work for the same organizations. To mitigate this risk, we sent up to three invites per organization.

6 RELATED WORK

Smart Contract Attacks and Security. To detect vulnerabilities in smart contracts, various tools using different techniques have been developed. Static analysis [5, 6, 16, 25, 45] is one such approach, where the source code or bytecode is analyzed without execution. In contrast, dynamic analysis examines the smart contract while executing it. Fuzzing [19, 23, 51] is a testing technique where inputs are automatically generated to test the system's behavior. Symbolic execution [11, 18, 30, 31] and formal verification [35, 43] are other well-known and frequently used techniques. However, formal verification typically requires users to provide specifications of intended behavior. In our study, we included at least one tool from each category that can be executed automatically, providing a comprehensive assessment of available solutions.

DeFi Attacks and Security. DeFi attacks present unique challenges compared to those in traditional financial systems, primarily due to two key factors [38, 48]: *(i)* the transparency in DeFi's application design, bytecode availability, and P2P transaction propagation; and *(ii)* the composability of DeFi applications. Several studies have examined DeFi attacks, including Zhou et al.'s five-layered framework for incident categorization and evaluation [61]. Other significant works have focused on specific security issues. For instance, the Flash Boys paper [12] was the first to explore the front-running issue, while Zhou et al. pioneered the study of sandwich attacks [60], which takes advantage of users' slippage settings in decentralised exchanges. DeFiRanger [50] extracted DeFi actions and identified price oracle manipulation attacks using pattern matching. DeFiPoser [59] employed SMT solvers to compose DeFi protocols, aiming to generate attacks. Collectively, these studies highlight the complexity and unique challenges posed by DeFi attacks. Additionally, our work underscores the limitations of traditional security tools that primarily focus on the smart contract layer neglecting the protocol layer.

Surveys on smart contract vulnerabilities and security tools. Atzei et al. [4] performed the first survey of smart contract attacks. Chen et al. [8] conducted a more comprehensive survey of 40 vulnerabilities, 29 attacks, and 51 defense locations and underlying causes, while Demolino et al. [13], categorized bugs based on typical developer pitfalls. Harz et al. [21] investigated 10 smart contract verification tools, exhibiting various aspects of their security characteristics. Hu et al. [22] assessed 39 analysis tools in terms of input type and methodology. Finally, Kushwaha et al. [27] presented a comprehensive survey of 86 analysis tools, the most of any research publication and article, and examined their analysis approaches and tool type. In constrast to these studies, we focus on the real-world impact of security tools by evaluating them against high-profile attacks and surveying practitioners.

Surveys of program analysis and security tools. Outside the realm of smart contracts security, Christakis et al. [10] empirically investigate what appeals to practitioners the most about a program analyzer [10], while [42] evaluates the usability of security tools. Johnson et al. [24] and Witschey et al. [49] explored why security tools are underused despite their benefits. On the contrary, in this work, we focus on how practitioners use security tools in the DeFi ecosystem. Finally, to the best of our knowledge, we are the first to survey auditors regarding security tool usage.

7 DATA AVAILABILITY

All the data and the analysis of this study will be accessible in an online repository upon the publication of this work.

8 CONCLUSIONS

In conclusion, our evaluation of automated security tools, combined with surveys of developers and auditors, reveals that existing tools have limited effectiveness in detecting high-impact vulnerabilities, with only 8% of the attacks in our dataset being detectable by automated tools. This indicates that smart contract and DeFi security has not been fully addressed yet. While reentrancy vulnerabilities can be detected, the tools do not adequately address logic-related bugs and protocol-layer vulnerabilities. We propose that researchers should prioritize the development of techniques that cover a wider range of vulnerabilities, including logic-related bugs, even if they partially require user input. Additionally, we suggest developing distinct tools for developers and auditors, as they have varying requirements regarding the capabilities of security tools. We hope that our findings can provide valuable insights and guidance for practitioners and researchers working in this dynamic and challenging area.

ACKNOWLEDGMENTS

We would like to express our sincere gratitude to all the participants who took the time to complete our surveys and provided invaluable feedback for our research. We also extend our thanks to Zhuo Zhang for the fruitful discussions on the impact of our results.

REFERENCES

[1] 2023. code4rena. https://code4rena.com/

[2] 2023. Defillama. https://defillama.com/

[3] 2023. Foundry's fuzzer. https://book.getfoundry.sh/forge/fuzz-testing

[4] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust*, Matteo Maffei and Mark Ryan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 164–186.

[5] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 454–469. https://doi.org/10.1145/3385412.3385990

[6] 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. https://doi.org/10.48550/ARXIV.1809.03981

[7] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, Yan Cai, and Zijiang Yang. 2019. sCompile: Critical Path Identification and Analysis for Smart Contracts. arXiv:1808.00624 [cs.CR]

[8] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2020. A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses. *ACM Comput. Surv.* 53, 3, Article 67 (jun 2020), 43 pages. https://doi.org/10.1145/3391195

[9] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2021. DefectChecker: Automated Smart Contract Defect Detection by Analyzing EVM Bytecode. *IEEE Transactions on Software Engineering* PP (01 2021), 1–1. https://doi.org/10.1109/TSE.2021.3054928

[10] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 332–343. https://doi.org/10.1145/2970276.2970347

[11] ConsenSys. [n. d.]. Consensys/mythril: Security Analysis Tool for EVM bytecode. supports smart contracts built for Ethereum, Hedera, quorum, Vechain, Roostock, Tron and other EVM-compatible blockchains. https://github.com/ConsenSys/mythril

[12] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2020. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 910–927.

[13] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. 2016. Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab, Vol. 9604. 79–94. https://doi.org/10.1007/978-3-662-53357-4_6

[14] 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* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 530–541. https://doi.org/10.1145/3377811.3380364

[15] Noama Fatima Samreen and Manar H. Alalfi. 2020. Reentrancy Vulnerability Identification in Ethereum Smart Contracts. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. 22–29. https://doi.org/10.1109/IWBOSE50093.2020.9050260

[16] 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.

[17] Robert Feldt and Ana Magazinius. 2010. Validity Threats in Empirical Software Engineering Research - An Initial Survey. 374–379.

[18] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. ETHBMC: A Bounded Model Checker for Smart Contracts. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2757–2774. https://www.usenix.org/conference/usenixsecurity20/presentation/frank

[19] Gustavo Grieco, Will Song, and Artur Cygan. [n. d.]. Echidna: Effective, usable, and fast fuzzing for smart contracts. https://agroce.github.io/issta20.pdf

[20] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. *A Semantic Framework for the Security Analysis of Ethereum Smart Contracts*. 243–269. https://doi.org/10.1007/978-3-319-89722-6_10

[21] Dominik Harz and William Knottenbelt. 2018. Towards safer smart contracts: A survey of languages and verification methods. https://arxiv.org/abs/1809.09805

[22] Bin Hu, Zongyang Zhang, Jianwei Liu, Yizhong Liu, Jiayuan Yin, Rongxing Lu, and Xiaodong Lin. 2021. A comprehensive survey on smart contract construction and execution: Paradigms, tools, and systems.

[23] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/3238147.3238177

[24] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 672–681.

[25] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *NDSS*.

[26] Barbara A Kitchenham and Shari L Pfleeger. 2008. Personal opinion surveys. *Guide to advanced empirical software engineering* (2008), 63–92.

[27] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. 2022. Ethereum Smart Contract Analysis Tools: A Systematic Review. *IEEE Access* 10 (2022), 57037–57062. https://doi.org/10.1109/ACCESS.2022.3169902

[28] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. ReGuard: Finding Reentrancy Bugs in Smart Contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 65–68. https://doi.org/10.1145/3183440.3183495

[29] Han Liu, Chao Liu, Wenqi Zhao, Yu Jiang, and Jiaguang Sun. 2018. S-gram: towards semantic-aware security auditing for ethereum smart contracts. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 814–819.

[30] 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* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

[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)*. 1186–1189. https://doi.org/10.1109/ASE.2019.00133

[32] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. SFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 778–788. https://doi.org/10.1145/3377811.3380334

[33] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th annual computer security applications conference*. 653–663.

[34] Daniel Perez and Benjamin Livshits. 2021. Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited. In *USENIX Security Symposium.*

[35] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1661–1677. https://doi.org/10.1109/SP40000.2020.00024

[36] Protofire. 2023. Protofire/solhint: Solhint is an open source project created by https://protofire.io. its goal is to provide a linting utility for solidity code. https://github.com/protofire/solhint

[37] Peng Qian, Zhenguang Liu, Qinming He, Roger Zimmermann, and Xun Wang. 2020. Towards Automated Reentrancy Detection for Smart Contracts Based on Sequential Models. *IEEE Access* 8 (2020), 19685–19695. https://doi.org/10.1109/ACCESS.2020.2969429

[38] Kaihua Qin, Liyi Zhou, Yaroslav Afonin, Ludovico Lazzaretti, and Arthur Gervais. 2021. CeFi vs. DeFi–Comparing Centralized to Decentralized Finance. *arXiv preprint arXiv:2106.08157* (2021).

[39] Meng Ren, Zijing Yin, Fuchen Ma, Zhenyang Xu, Yu Jiang, Chengnian Sun, Huizhong Li, and Yan Cai. 2021. 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* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 566–579. https://doi.org/10.1145/3460319.3464837

[40] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2018. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934* (2018).

[41] Edward Smith, Robert Loftin, Emerson Murphy-Hill, Christian Bird, and Thomas Zimmermann. 2013. Improving developer participation rates in surveys. In *6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. 89–92.

[42] Justin Smith, Lisa Nguyen Quang Do, and Emerson Murphy-Hill. 2020. Why Can't Johnny Fix Vulnerabilities: A Usability Evaluation of Static Analysis Tools for Security. In *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*. USENIX Association, 221–238. https://www.usenix.org/conference/soups2020/presentation/smith

[43] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. 2019. VeriSmart: A Highly Precise Safety Verifier for Ethereum Smart Contracts. arXiv:1908.11227 [cs.PL]

[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)*. 103–119. https://doi.org/10.1109/EuroSP51992.2021.00018

[45] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 67–82. https://doi.org/10.1145/3243734.3243780

[46] Zhiyuan Wan, Xin Xia, David Lo, Jiachi Chen, Xiapu Luo, and Xiaohu Yang. 2021. Smart Contract Security: A Practitioners' Perspective: The Artifact of a Paper Accepted in the 43rd IEEE/ACM International Conference on Software Engineering (ICSE 2021). In *Proceedings of the 43rd International Conference on Software Engineering: Companion Proceedings* (Virtual Event, Spain) *(ICSE '21)*. IEEE Press, 227–228. https://doi.org/10.1109/ICSE-Companion52605.2021.00104

[47] Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. 2021. ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts. *IEEE Transactions on Network Science and Engineering* 8, 2 (2021), 1133–1144. https://doi.org/10.1109/TNSE.2020.2968505

[48] Sam M. Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William J. Knottenbelt. 2022. SoK: Decentralized Finance (DeFi). arXiv:2101.08778 [cs.CR]

[49] Jim Witschey, Olga Zielinska, Allaire Welk, Emerson Murphy-Hill, Chris Mayhorn, and Thomas Zimmermann. 2015. Quantifying Developers' Adoption of Security Tools. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 260–271. https://doi.org/10.1145/2786805.2786816

[50] Siwei Wu, Dabao Wang, Jianting He, Yajin Zhou, Lei Wu, Xingliang Yuan, Qinming He, and Kui Ren. 2021. DeFiRanger: Detecting Price Manipulation Attacks on DeFi Applications. arXiv:2104.15068 [cs.CR]

[51] Valentin Wüstholz and Maria Christakis. 2020. Harvey: A Greybox Fuzzer for Smart Contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1398–1409. https://doi.org/10.1145/3368089.3417064

[52] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. 2021. Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1029–1040. https://doi.org/10.1145/3324884.3416553

[53] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. 2022. xFuzz: Machine Learning Guided Cross-Contract Fuzzing. *IEEE Transactions on Dependable and Secure Computing* (2022).

[54] Jiaming Ye, Mingliang Ma, Yun Lin, Yulei Sui, and Yinxing Xue. 2020. Clairvoyance: Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 274–275. https://doi.org/10.1145/3377812.3390908

[55] Pengcheng Zhang, Feng Xiao, and Xiapu Luo. 2019. SolidityCheck : Quickly Detecting Smart Contract Problems Through Regular Expressions. arXiv:1911.09425 [cs.SE]

[56] Pengcheng Zhang, Feng Xiao, and Xiapu Luo. 2020. A Framework and DataSet for Bugs in Ethereum Smart Contracts. 139–150. https://doi.org/10.1109/ICSME46990.2020.00023

[57] Zhuo Zhang, Brian Zhang, Xu Wen, and Zhiqiang Lin. 2023. Demystifying smart contract vulnerabilities. In *ICSE*.

[58] Ence Zhou, Song Hua, Bingfeng Pi, Jun Sun, Yashihide Nomura, Kazuhiro Yamashita, and Hidetoshi Kurihara. 2018. Security Assurance for Smart Contract. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. 1–5. https://doi.org/10.1109/NTMS.2018.8328743

[59] Liyi Zhou, Kaihua Qin, Antoine Cully, Benjamin Livshits, and Arthur Gervais. 2021. On the just-in-time discovery of profit-generating transactions in defi protocols. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 919–936.

[60] Liyi Zhou, Kaihua Qin, Christof Ferreira Torres, Duc V Le, and Arthur Gervais. 2021. High-frequency trading on decentralized on-chain exchanges. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 428–445.

[61] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. 2022. SoK: Decentralized Finance (DeFi) Attacks. Cryptology ePrint Archive, Paper 2022/1773. https://eprint.iacr.org/2022/1773 https://eprint.iacr.org/2022/1773.