Static Application Security Testing (SAST) Tools for Smart Contracts: How Far Are We?

KAIXUAN LI, East China Normal University, China
YUE XUE, MetaTrust Labs, Singapore
SEN CHEN[*], Tianjin University, China
HAN LIU, East China Normal University, China
KAIRAN SUN, Nanyang Technological University, Singapore
MING HU, Nanyang Technological University, Singapore
HAIJUN WANG, Xi'an Jiaotong University, China
YANG LIU, Nanyang Technological University, Singapore
YIXIANG CHEN, East China Normal University, China

In recent years, the importance of smart contract security has been heightened by the increasing number of attacks against them. To address this issue, a multitude of static application security testing (SAST) tools have been proposed for detecting vulnerabilities in smart contracts. However, objectively comparing these tools to determine their effectiveness remains challenging. Existing studies often fall short due to the taxonomies and benchmarks only covering a coarse and potentially outdated set of vulnerability types, which leads to evaluations that are not entirely comprehensive and may display bias.

In this paper, we fill this gap by proposing an up-to-date and fine-grained taxonomy that includes 45 unique vulnerability types for smart contracts. Taking it as a baseline, we develop an extensive benchmark that covers 40 distinct types and includes a diverse range of code characteristics, vulnerability patterns, and application scenarios. Based on them, we evaluated 8 SAST tools using this benchmark, which comprises 788 smart contract files and 10,394 vulnerabilities. Our results reveal that the existing SAST tools fail to detect around 50% of vulnerabilities in our benchmark and suffer from high false positives, with precision not surpassing 10%. We also discover that by combining the results of multiple tools, the false negative rate can be reduced effectively, at the expense of flagging 36.77 percentage points more functions. Nevertheless, many vulnerabilities, especially those beyond Access Control and Reentrancy vulnerabilities, remain undetected. We finally highlight the valuable insights from our study, hoping to provide guidance on tool development, enhancement, evaluation, and selection for developers, researchers, and practitioners.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**; • **Security and privacy** → **Software security engineering**.

---

[*]Corresponding author.

Authors' Contact Information: Kaixuan Li, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China, kaixuanli@stu.ecnu.edu.cn; Yue Xue, MetaTrust Labs, Singapore, Singapore, xueyue@metatrust.io; Sen Chen, College of Intelligence and Computing, Tianjin University, Tianjin, China, senchen@tju.edu.cn; Han Liu, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China, hanliu@stu.ecnu.edu.cn; Kairan Sun, Nanyang Technological University, Singapore, Singapore, sunk0013@e.ntu.edu.sg; Ming Hu, Nanyang Technological University, Singapore, Singapore, ecnu_hm@163.com; Haijun Wang, Xi'an Jiaotong University, Xi'an, China, haijunwang@xjtu.edu.cn; Yang Liu, Nanyang Technological University, Singapore, Singapore, yangliu@ntu.edu.sg; Yixiang Chen, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China, yxchen@sei.ecnu.edu.cn.

---

1 INTRODUCTION

As smart contracts gain prominence in the blockchain technology landscape, their secure and efficient implementation becomes increasingly crucial. These self-executing digital agreements automate transactions and enforce contract terms, but they are not immune to vulnerabilities and errors. It was reported that security vulnerabilities in smart contracts have resulted in more than $4.75 billion in financial losses from 2012 to 2022 [17]. Therefore, ensuring the reliability and security of smart contracts is essential to safeguard the trustworthiness of decentralized applications.

To this end, a wide array of techniques have been proposed to detect vulnerabilities such as integer overflow/underflow. These approaches can be broadly classified into categories including static application security testing (SAST) [19, 23, 36, 66, 67, 69] performing analysis without running the programs and dynamic application security testing (DAST) [15, 29], which involves testing the contracts with various inputs either in simulated environments before deployment or by executing the code post-deployment. Given the immutable nature of smart contracts, early vulnerability detection is of paramount importance [3]. Compared with DAST, SAST provides more immediate and comprehensive insights, particularly during the coding phases.

It is necessary to clearly understand the detection capability of current existing SAST tools for different stakeholders such as tool developers, researchers, and practitioners. However, it still remains challenging to objectively compare these SAST tools to determine their effectiveness. Existing studies [20, 23, 25, 66] have demonstrated the effectiveness of some tools through a series of evaluations on their own experimental scenarios, but their limitations on evaluation settings: *1)* **Lack of an up-to-date and fine-grained taxonomy.** Taxonomies [20, 25, 44, 66, 76] are typically employed before the comparison and evaluation of SAST tools, which contributes to constructing benchmarks and testing tools. Nonetheless, existing taxonomies in [13, 20, 25, 38, 76] exhibit limitations in an outdated and coarse classification of vulnerabilities. For example, Durieux [20] noted that the state-of-the-art DASP Top 10 [38] taxonomy, comprising 10 coarse vulnerability types, might not sufficiently cover all vulnerabilities affecting smart contracts. As for Smart Contract Weakness Classification (SWC) [65], despite comprising 37 entries, a considerable portion is primarily associated with code quality issues, such as *SWC-103: Floating Pragma*. Meanwhile, DASP Top 10, proposed in 2016 and last updated in 2018, includes vulnerabilities including *Short Addresses* that are now obsolete due to the introduction of corresponding security countermeasures. Additionally, existing taxonomies suffer from ambiguous classification and unreasonable granularity. Concretely, there are overlaps in DASP Top 10 between types, leading to ambiguities in classification (see § 3.2). *2)* **Lack of a comprehensive benchmark.** The benchmarks used in existing studies [20, 25] suffer from issues such as small sizes or limited vulnerability types. The use of these non-comprehensive benchmarks may result in varying evaluation conclusions. Ren et al. [44] concluded that evaluation results are highly reliant on benchmark suites, causing the effectiveness of Slither [23] and SmartCheck [66] to vary across three distinct benchmarks due to the limitations in vulnerability types and code characteristics within these benchmarks. These limitations within benchmarks contribute to evaluation conclusions that may be neither objective nor fair. Additionally, due to the limitations within their benchmarks, while most related work [20, 25] used evaluation metrics such as Precision or Recall, they generally evaluated only partial metrics, which also hinders objective evaluation conclusions.
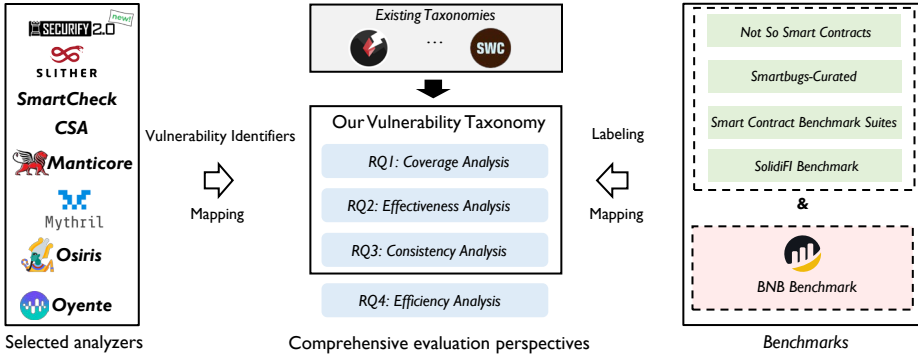
Fig. 1. Overview of our study.

Therefore, as shown in Figure 1, to tackle the above-mentioned challenges and problems, we first developed an up-to-date and fine-grained vulnerability taxonomy for smart contracts, including 45 unique types, which is based on existing taxonomies including DASP Top 10, SWC, the vulnerability identifiers used in SAST tools, as well as our domain knowledge. This unified and up-to-date taxonomy facilitates the construction of our benchmark. We further collected and constructed a systematic and diverse benchmark suite by collecting existing high-quality datasets including [20, 25, 44, 68]. Meanwhile, to enhance the benchmark's diversity and provide a more realistic representation of real-world projects, we also collected 2,941 representative industrial smart contract projects from BscScan [8].

Notably, since some of these datasets only label partial vulnerabilities or do not disclose labeling information, we took substantial efforts (11 person-months) to manually label the ground truth by involving three security auditors to label and map them to our taxonomy at the function level. In total, our benchmark includes 788 smart contract files and 10,394 vulnerabilities (ground truth), which also covers almost all vulnerability types in our proposed taxonomy and includes a wide range of code characteristics, vulnerability patterns, and application scenarios.

Based on the new taxonomy and benchmark, we selected 8 representative SAST tools for smart contracts to compare and evaluate them. Concretely, *1)* to explore and observe the detection coverage of each tool, we mapped their vulnerability identifiers to our proposed taxonomy. *2)* To investigate the effectiveness of these tools, we conducted a benchmark experiment on our benchmark by adopting more fair runtime parameters and evaluation metrics, i.e., Recall, Precision, and F1-score. *3)* We also performed a consistency evaluation to observe the potential for combining multiple tools. *4)* Finally, we evaluated the efficiency of these tools by observing their time cost in analyzing all of the smart contracts in our benchmark.

Through our study, we have discovered the following key findings: *1)* CSA[1] and Securify2 display higher vulnerability coverage than other tools. *2)* Regarding effectiveness, CSA maintains its top position in both recall and precision, while Slither's performance drops due to a higher false positive rate. However, the existing tools still miss around 50% cases in our benchmark, with precision not surpassing 10%. *3)* For the consistency analysis, tool combination can effectively reduce the false negatives to 29.3%, at the expense of flagging 36.77 percentage points more functions. Additionally, vulnerabilities related to *Access Control* and *Reentrancy* are generally easier for tools to detect than those in the *Arithmetic* category. *4)* Our efficiency analysis shows that tools using

---

[1]The anonymous commercial static analysis tool we evaluated.

symbolic execution take more time compared to those with static analysis techniques. Specifically, Manticore takes the longest time for analysis, and Securify2 demands more memory resources, while SmartCheck emerges as the fastest tool among those evaluated.

In summary, our main contributions are as follows:

- We proposed a new vulnerability taxonomy for smart contracts, based on existing taxonomies, the vulnerability identifiers used in SAST tools, and our domain knowledge.
- We constructed a comprehensive and diverse benchmark of 788 smart contracts, uncovering 10, 394 vulnerabilities at the function level. The meticulous effort to establish this ground truth took *11 person-months*, making it the largest benchmark dedicated to smart contract vulnerabilities to date.
- We conducted a benchmark experiment of 8 SAST tools using our diverse benchmark and performed a large-scale experiment on 8,981 smart contracts (i.e., $8 \times (788 + 8, 981) = 78, 152$ scanning tasks). This preparation and execution process took over 4 months. We performed an extensive and rigorous empirical evaluation of 8 existing SAST tools from a comprehensive perspective including coverage, effectiveness, consistency, and efficiency. The study data and code are released on the website: https://sites.google.com/view/sc-sast-study-fse2024/home.

2   RELATED WORK

2.1   Empirical Study on SAST Tools for Smart Contracts

There has been extensive research conducted to evaluate security tools for smart contracts. Ghaleb and Pattabiraman [25] proposed an automated approach, SolidiFI, to evaluate static analyzers for smart contracts by injecting code defects into 50 smart contracts, introducing 9,369 security vulnerabilities, and testing the generated buggy contracts using six static analyzers. Rameder [43] conducted a systematic literature and tool review, providing a comprehensive overview of tools, classifications of smart contract vulnerabilities, and detection methods including fuzzing, formal methods, and static analysis. Durieux et al. [20] presented an empirical evaluation of nine analyzers based on 69 annotated vulnerable smart contracts and another 47,518 contracts without ground truth. However, our work aims to conduct a systematic evaluation of static analyzers by constructing a more diverse benchmark (ground truth) and evaluating them from multi-dimensional perspectives including coverage and granularity, effectiveness, consistency, and efficiency. Ren et al. [44] proposed a unified standard that includes a 4-step evaluation process to eliminate bias in the assessment process. They evaluated nine representative tools based on 46,186 source-available smart contracts collected from four influential organizations. While they emphasized that experiment setups can lead to different or even incorrect conclusions during the evaluation, they did not systematically compare the detection capacities of the tools. Monteiro [35] introduced SmartBugs, an extendable execution framework that includes 47,661 Solidity smart contracts, and presented an evaluation of seven state-of-the-art tools. The tool evaluation framework they proposed contributes to our research on running security tools for Solidity smart contracts. However, the taxonomies they used are outdated or incomplete, which will be discussed in § 3.2.

Another related work was proposed by Chen et al. [13], in which they collected and analyzed smart-contract-related posts and real-world smart contracts, defining 20 types of contract defects that impact smart contract quality. However, they did not compare or evaluate any security tools. We do not merely construct a systematic taxonomy for smart contract vulnerabilities and further collect a comprehensive benchmark, but evaluate the performance of eight static analyzers based on the diverse vulnerability types. Zhou et al. [77] constructed a DeFi reference framework that categorizes 77 academic papers, 30 audit reports, and 181 incidents, revealing differences between the academic and practitioner communities in defending and examining incidents. Their results

show that DeFi security is still in its infancy and that many potential defense mechanisms require further research and implementation. Meanwhile, Chaliasos [11] also focused on DeFi attacks, evaluating automated security tools and surveying 49 smart contract developers and auditors. They underlined the limited effectiveness of current tools in detecting high-impact vulnerabilities, with only 8% of the attacks in their dataset being detected by automated tools. This emphasizes that smart contracts and DeFi security still have significant room for improvement. While reentrancy vulnerabilities can be detected, existing tools struggle to address logic-related bugs and protocol-layer vulnerabilities. Contrasting their work, our study focuses on vulnerabilities beyond DeFi attacks, as DeFi vulnerabilities are typically business-logic-related and considered too challenging for static analyzers to detect effectively. Akca et al. [1] evaluated various techniques for testing smart contracts including fuzzing and genetic algorithm rather than static analysis.

Our work distinguishes itself by focusing on conducting a systematic evaluation of SAST tools for Solidity smart contracts, constructing a comprehensive benchmark, and evaluating the tools from various perspectives. We aim to provide valuable insights into the performance of these tools and contribute to the ongoing development and improvement of SAST tools in the context of smart contract security. In summary, our work differs from the state of the art in terms of the considered **(1) vulnerability taxonomies** (an up-to-date and fine-grained vulnerability taxonomy for smart contracts), **(2) evaluation benchmarks** (the largest benchmark covering comprehensive and diverse vulnerability types), **(3) evaluation methodology** (mapping detection rules and ground truth to our taxonomy), **(4) Finer detection code granularity** (vulnerable *function-level*), and **(5) evaluation perspectives** (coverage analysis on supported vulnerabilities, effectiveness analysis on vulnerability detection, consistency analysis among tools' focuses, and efficiency analysis).

2.2 SAST Tools for Smart Contracts

Several works focused on detecting vulnerabilities in smart contracts [6, 19, 23, 26, 34, 36, 66, 67, 69], with some specifically targeting certain types of vulnerabilities. Apart from the selected tools we evaluated, Liao et.al [32] proposed a framework named SmartDagger to detect cross-contract vulnerability through static analysis at the bytecode level. Besides, Liu et al. [34] proposed mining past transactions of a contract to recover a probable access control model to identify potential user permission-related bugs. They implemented their role mining and security policy validation in a tool called SPCon. Similarly, Ghaleb et al. [26] introduced AChecker to detect permission-related vulnerabilities by combining data-flow analysis and symbolic execution techniques. Recently, Fang et al. [22] developed SoMo which leverages symbolic execution to detect `modifier` issues.

3 OVERVIEW

In this section, we will introduce the overview of our study design including tool selection (§ 3.1), taxonomy construction (§ 3.2), dataset collection (§ 3.3), and research questions (§ 3.4), as also shown in Figure 1.

3.1 Tool Selection

To collect representative SAST tool candidates for smart contracts, we conducted a systematic literature review (SLR) as follows. *1)* We first used several keywords such as "smart contract", "static analysis", and "security tool" to search papers published in top-tier Software Engineering, Security, and Programming Language venues in the last three years (till Sep. 2023), including FSE, ICSE, S&P, Usenix Security, PLDI, POPL, etc. We obtained 59 research papers related to security for smart contracts as an initial paper list. *2)* By further excluding papers that are unrelated to SAST tools, we thereby got 14 papers related. We gathered a list of 44 SAST tools. Our selection process was structured to ensure the relevance and practicality of the tools for our analysis.

Table 1. List of selected SAST tools. # Baseline: Tool usage as a baseline in 2021-2023. # Stars: GitHub repository stars. # Citation: Number of citations.

Technology	Tool	Analysis Level	# Baseline	# Citation	# Stars	Publication
Static Analysis	**Securify2**	Source Code	7	649	529	CCS'18[69]
	Slither	Source Code	7	244	**4.5k**	WETSEB'19[23]
	SmartCheck	Source Code	7	481	315	WETSEB'18[66]
	CSA	Source Code	/	/	/	/
Symbolic Execution	**Manticore**	Bytecode	7	185	3.5k	ASE'19[36]
	Mythril	Bytecode	11	/	3.5k	White Paper
	Osiris	Bytecode	6	222	50	ACSAC'18[67]
	Oyente	Bytecode	8	**1897**	1.3k	CCS'16[19]

Criterion #1 (Availability): The tool should be publicly accessible. Hence, we excluded 16 tools due to unavailability (commercial or closed-source tools). Notable exclusions in this step included tools like Zeus [30], Sereum [46], and MythX [37].

Criterion #2 (Security related): We try to select tools that identify security vulnerabilities, rather than those aimed at detecting code quality issues, we thereby filtered out 2 linters including Solhint [60] and Ethlint [21].

Criterion #3 (Generalized SAST tools): We then narrowed our focus to "generalized" SAST tools since we aim to compare and evaluate tools across various vulnerability types (45 unique ones in this study). It required the tools to support at least **five** vulnerability types rather than specializing in a few vulnerability types, such as only aiming at few vulnerabilities within access control (e.g., AChecker [26]), DoS (e.g., eTainter [27]), state inconsistency (e.g., SailFish [6]), etc. After this filtering, we retained 16 tools but excluded 10 tools. Note that SailFish [6] was excluded since it only supports 3 vulnerability types.

Criterion #4 (CLI & no additional input need): Next, due to the need for large-scale analysis, we focused on tools that support CLI and accept source code directly. Based on it, we excluded 4 tools including teEther [31], SmarTest [58], NPChecker [70], and SmartPulse [62] from our list since they required additional inputs like specifications, transactions, or test cases.

Criterion #5 (Popularity and relevance): In the final step, we considered several quantitative metrics that reflect their impact, adoption, and relevance in the research community and among practitioners. Specifically, we investigated ① the frequency of employed as baselines in relevant studies over the recent three years (# Baseline), ② the citations they receive (# Citation), and ③ the number of stars in their GitHub repositories (# Stars) till September 2023. We thereby got 7 tools that are both more popular and relevant in the field.

According to the unified criteria above, we finally got 7 open-source SAST tools: Securify2 [10], Slither [23], SmartCheck [66], Manticore [36], Mythril [16], Osiris [67], and Oyente [19] (see Table 1). The detailed process of each step can be accessed on our website [72]. Meanwhile, to observe the gap in effectiveness between open-source and commercial tools, we try to include commercial tools for smart contracts. However, due to the significant financial expenses, it is not feasible to run multiple commercial tools on our extensive dataset (8,981 contract files). We finally successfully obtained access to a commercial static analysis tool (CSA) through our industrial partner.

Selected tools. As displayed in Table 1, the selected tools leverage state-of-the-art analysis techniques and were frequently compared or evaluated in recent works [6, 20, 22, 25, 26, 30, 44, 76] and/or are popular among practitioners [12].

Securify2 (★ 529) is a successor for Securify security scanner [9, 69], while the latter is deprecated since 2020. In our study, we use version v2.0.

Slither (★ 4.5k) is a static analysis framework developed by Trail of Bits [39]. It converts Solidity smart contracts into an intermediate representation (IR) called SlithIR and applies Static Single Assignment (SSA) [2, 18, 47] to perform data flow analysis, as well as taint tracking to extract and refine information. In our study, we use version v0.9.3.

SmartCheck (★ 315) is a static analysis tool developed by SmartDec [57]. It uses lexical and syntactical analysis performed on Solidity source code to look for vulnerabilities. We use v2.0 of SmartCheck here.

CSA refers to (Commercial Static Analysis), which uses static analysis technique to find vulnerabilities in smart contracts and is the commercial tool used in this study. Similar to the other tools, we make sure to use one of the latest versions of CSA. To protect the company behind this tool, we anonymize its name and neither reveal its exact version and release date nor the implemented analysis technique.

Manticore (★ 3.5k) is a symbolic execution tool for smart contracts. And it is integrated with Z3 [45] which is a powerful theorem prover and solver for satisfiability modulo theories (SMT) [4] problems. We use version v0.3.7 here.

Mythril (★ 3.5k) is developed by ConsenSys [37]. It performs symbolic execution for smart contracts, relying on taint analysis, and control flow analysis of the EVM bytecode to prune the search space and to look for values that allow exploiting vulnerabilities. We use version v0.23.15.

Osiris (★ 50) was developed based on Oyente, which detects vulnerabilities in Ethereum smart contracts. Here, we use version #d1ecc37.

Oyente (★ 1.3k) is the earliest academic tool in this domain, and it is still continuously used in evaluations of numerous academic work [12, 14, 20, 25, 26, 29, 33, 42, 44, 50, 73, 76]. It performs symbolic execution for vulnerability detection. And we use version v0.2.7 here.

3.2 Taxonomy Construction

Many taxonomies have been proposed [11, 38, 65, 76] to facilitate vulnerability understanding, such as DASP Top 10 and SWC. However, they exhibit several limitations: ① Outdated and coarse types: Existing taxonomies may lack newly discovered vulnerability types or retain outdated types. For instance, DASP Top 10 [38], used by Durieux [20], was proposed in 2016 and last updated in 2018, leading to outdated types such as the "Short Addresses" issue, which has been resolved by the Solidity compiler from v0.5.0. Moreover, SWC was established in 2017 and contains only 37 types of weaknesses, many of which focus on code quality rather than severe security risks. ② Ambiguous classification and unreasonable granularity: Some taxonomies exhibit overlaps between categories, leading to ambiguities in classification. For example, the taxonomy used by [38] shows overlaps between categories such as *Access Control* and *Unchecked Low-Level Calls*, which might result in confusion when categorizing vulnerabilities. Furthermore, the limitations in existing taxonomies make it challenging to establish a reliable benchmark for evaluating the actual performance of SAST tools or techniques in addressing smart contract vulnerabilities. For instance, the datasets used in [20, 25], and [35] are limited in benchmark sizes or vulnerability types, which lead to incomparable results and hinder the evaluation of the tools' effectiveness in vulnerability detection.

Hence, an up-to-date and fine-grained taxonomy of known smart contract vulnerabilities is essential. This foundation allows for more objective and reliable comparison and evaluation of tools in the domain. To address this, we constructed a new vulnerability taxonomy for smart contracts from five perspectives as follows: ① We began by addressing the outdated nature of existing taxonomies, such as the DASP Top 10 and SWC. We removed outdated categories like the *Short Addresses* and added newly discovered vulnerability categories *Storage and Memory* to ensure a comprehensive representation of known smart contract vulnerabilities. ② We reorganized the taxonomy by analyzing the root causes of vulnerabilities, such as those in *Block Manipulation*

and *Cryptographic*, to create a more structured approach in line with the ETSI smart contract standard. ③ To resolve ambiguity and granularity issues in existing taxonomies, we combined overlapping categories and carefully analyzed their parent-child relationships. This step was to diminish ambiguity and redundancy, ensuring a clear and concise classification of vulnerabilities. ④ Meanwhile, to ensure our taxonomy's relevance and practicality, we also considered the supported vulnerability types among state-of-the-art SAST tools by examining their vulnerability identifiers and aligning them with the categories in our taxonomy. This step was crucial to guarantee the applicability of our taxonomy to real-world tools and practices. ⑤ Lastly, since our primary concern is vulnerabilities in smart contracts, we removed entries related solely to code quality.

In summary, as shown in Table 2, our taxonomy extends and refines the existing taxonomies including DASP Top 10 and SWC by addressing their limitations and reorganizing the structure for a more precise representation of smart contract vulnerabilities. In total, our proposed taxonomy encompasses 45 vulnerability types, organized into 7 distinct categories. Due to space limitations, details on our taxonomy can be accessed on our website [71].

3.3 Dataset Collection

3.3.1 Benchmark Construction. To create a diverse benchmark covering as many vulnerability types as possible from our taxonomy for a thorough SAST tools evaluation, we defined three criteria for collecting datasets as follows:

Criterion #1 (Solidity smart contracts and availability): The benchmark must consist of open-source Solidity smart contracts. Since our evaluation focuses on tools specifically designed for Solidity, it is essential to collect appropriate language datasets that are readily available. We identified 19 open-source datasets that fulfill this requirement.

Criterion #2 (Popularity or peer-reviewed): The benchmark should be either widely used or peer-reviewed. This ensures that the dataset has been recognized and applied in related research papers, demonstrating its relevance and validity in the research community. We thereby included 7 benchmarks including [13, 25, 44, 55, 56, 68, 76].

Criterion #3 (Labeled vulnerability types): This criterion covers two aspects: ① The benchmark should primarily focus on vulnerabilities, and ② the ground truth should be labeled at least at the file level. As we aim to gather high-quality datasets and further label them at the function level, we excluded 3 benchmarks with no ground truth or labeled only at the project level, such as [13, 55, 76]. Finally, we obtained 4 available benchmarks [25, 44, 56, 68].

Furthermore, we tried to collect more real-world vulnerable smart contracts. To this end, we focused on selecting representative BNB projects from BscScan [8]. To obtain a representative sample, we first gathered the top 3,000 BNB projects ranked by their *Market Capitalization (Market Cap)* and *Liquidity Value (Liquidity)* as indicators. *Market Cap* reflects a project's total market value, indicating its prominence and adoption. *Liquidity* reflects the ease of trading a project's token, with higher values suggesting more accessible and stable markets. By selecting projects with high *Market Cap* and *Liquidity*, we ensured our dataset includes prominent and widely-used BNB contracts. After filtering out unavailable contracts, we obtained 2,941 addresses corresponding to 2,941 unique projects. We eventually got 2,941 representative BNB projects containing 8,249 smart contract files, focusing specifically on those with a *Market Cap* greater than $3,000 and the average *Liquidity* of $1,632,386, respectively. After this process, we collaborated with our industrial partner by engaging 3 security audit experts to construct the ground truth. We identified 120 vulnerabilities affecting 113 contracts from these BNB projects (function level) .

To remove duplicates from all five datasets within our benchmark suite, we checked the MD5 checksums of the contract files after eliminating blank lines and comments, a method commonly used in related studies [20, 63]. In the end, we obtained 8,981 unique contract files. Among these,

Table 2. Our proposed vulnerability taxonomy for smart contracts (# Types indicates the number of types in each category. "*" indicates types not included in our benchmark. A "√" indicates support by tools for a given type, followed by the number of detected samples from our benchmark. # Samples indicates the total count of vulnerability samples for each type within our benchmark. # SWC: the number of SWC entries mapped. # Supported: the number of types mapped.).

Category (# Types)	Type	Securify2	Slither	SmartCheck	CSA	Manticore	Mythril	Osiris	Oyente	# Samples
Access Control (18)	Arbitrary from in transferFrom() without msg.sender Check		√(11)		√(15)					17
	Call to Arbitrary Addresses with Unchecked Call data		√(1)		√(1)					1
	Caller Not Checked	√(2)	√(5)		√(10)					11
	Logic Contract Could be Destructed*		√		√					-
	Dangerous Immediate Initialization of State Variables		√(10)		√(9)					12
	Dangerous Usage of tx.origin	√(905)	√(1,169)		√(1,271)		√(244)			1,371
	Default Function Visibility		√(24)		√(20)		√(2)			29
	Initializing Method without Permission Check	√(0)	√(0)		√(1)					5
	Method permit() Used for Arbitrary from in transferFrom()	√(0)	√(1)		√(2)					6
	Missing msg.sender Check for transferFrom()	√(0)	√(0)		√(2)					7
	Missing Input Validation (Call Inject)	√(0)	√(13)		√(12)		√(0)			24
	Sending Ether to Arbitrary Destinations	√(0)	√(26)		√(46)		√(0)			56
	Unprotected Contract Destruction	√(0)	√(4)		√(16)	√(0)	√(2)	√(1)	√(1)	19
	Unprotected Ether Withdrawal	√(2)	√(36)	√(40)	√(48)	√(0)	√(16)			67
	Unsafe Delegatecall	√(0)	√(3)		√(0)	√(0)	√(0)			8
	Unused Return Value	√(0)	√(0)	√(0)	√(3)	√(0)	√(0)			6
	Usage of Public Mint or Burn	√(0)	√(1)	√(3)	√(6)					6
	Write to Arbitrary Storage Location	√(0)	√(2)	√(0)	√(2)		√(0)			6
Arithmetic (3)	Inappropriate Integer Division before Multiplication	√(0)	√(8)		√(2)		√(0)	√(1)		10
	Integer Overflow/Underflow	√(0)		√(1)	√(645)	√(0)	√(44)	√(325)	√(91)	1,976
	Unsafe Array Length Assignment	√(0)	√(2)	√(0)	√(4)					6
Block Manipulation (3)	Dangerous Usage of block.timestamp	√(370)	√(1,108)		√(1,132)	√(0)	√(210)	√(4)	√(0)	1,139
	Transaction Order Dependency	√(1,231)		√(479)	√(1,970)		√(0)	√(0)	√(0)	2,687
	Weak PRNG (Pseudorandom Number Generator)	√(1)	√(15)	√(1)	√(23)	√(0)				60
Cryptographic (2)	Lack of Proper Signature Verification	√(0)	√(0)		√(1)					1
	Signature Malleability		√(1)							1
Denial of Services (6)	transfer() and send() with Hardcoded Gas Amount	√(0)	√(0)	√(7)	√(0)					4
	Contract Could Lock Ether	√(50)	√(69)		√(82)					92
	DoS with Block Gas Limit	√(0)	√(0)		√(0)					8
	DoS With Failed Call	√(0)	√(0)		√(10)		√(2)	√(0)	√(0)	15
	Force Sending Ether with this.balance Check	√(0)	√(0)		√(0)					1
	Unsafe send() in the require() Condition	√(833)	√(0)	√(8)	√(743)	√(0)	√(0)			1,269
Reentrancy (5)	Reentrancy Vulnerability with Negative Events	√(13)	√(111)		√(119)					124
	Reentrancy Vulnerability with Transfer	√(37)	√(201)	√(201)	√(227)	√(0)	√(45)	√(20)	√(5)	227
	Reentrancy Vulnerability with Same Effect	√(69)	√(276)		√(297)					331
	Reentrancy Vulnerability with Token Transfer	√(83)	√(450)		√(513)	√(0)	√(53)	√(27)	√(27)	525
	Reentrancy Vulnerability without Token Transfer	√(34)	√(190)		√(218)					227
Storage & Memory (8)	Arbitrary Function Jump via Inline Assembly*		√							-
	Bytes Variables Risk*		√		√					-
	Dangerous Usage of msg.value inside a Loop*	√	√		√					-
	Error-prone Assembly Usage	√(0)	√(0)		√(0)	√(0)				1
	Memory Manipulation	√(0)	√(0)		√(1)					2
	Modifying Storage Array by Value*	√	√		√					-
	Payable Functions Using delegatecall inside a Loop	√(0)	√(0)		√(1)					2
	Uninitialized variable	√(0)	√(23)	√(0)	√(31)					35
	# SWC	16	17	8	**32**	8	14	8	6	-
	# Supported	37	43	14	**44**	11	17	8	7	-

there are 788 vulnerable contract files containing 10,394 vulnerabilities at the function level (refer to Table 2 and Table 3 for details).

Table 3. The composition of our benchmark. MI denotes manually injected vulnerabilities and RW for real-world vulnerabilities.

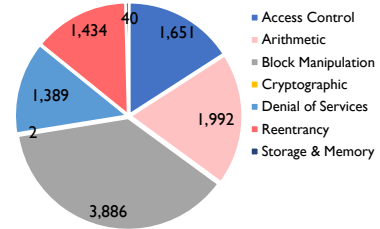| Datasets	# Files	# Vulnerabilities		Source
		MI	RW	
Not So Smart Contracts	18	✗	18	Trail of Bits [68]
Smartbugs-Curated	143	✗	257	ICSE'20 [56]
SolidiFI Benchmark	300	9,606	✗	ISSTA'20 [25]
Smart Contract Benchmark Suites	214	✗	393	ISSTA'21 [44]
BNB Benchmark	113	✗	120	BscScan [8]

Fig. 2. Distribution of vulnerabilities by category in our benchmark.

3.3.2 Mapping Vulnerability Identifiers and Benchmark Data to Our Taxonomy. To automatically evaluate these tools, one challenge we faced was that SAST tools use different vulnerability identifiers for their supported vulnerability types. For example, Mythril employs SWC entries in its reported issues, while others introduce their own vulnerability identifiers. This disparity makes it difficult to automatically determine whether a SAST tool detected a specific vulnerability type. Hence, we tried to map their vulnerability identifiers to the unified vulnerability types in our taxonomy. This required a thorough review of the vulnerability identifiers, descriptions, and associated source code provided by each tool. Furthermore, although the four datasets we collected have already classified the vulnerability data, the vulnerability categories and types differ since they use different taxonomies. Additionally, some of them [44, 68] specify the vulnerability location at the file level. To conduct a much fairer evaluation, we relabeled and remapped these vulnerability ground truths to our proposed vulnerability taxonomy at the function level. To ensure accuracy and minimize potential noise during this phase, we involved three security auditing experts for manual cross-validation of the data. In total, it took us 7.5 person-months to relabel and map the entire benchmark. An additional 3.5 person-months were spent on mapping vulnerability identifiers and cross-validating the mapping results (details in § 5.2.2).

As displayed in Table 2 and Figure 2, our benchmark has a nearly complete vulnerability type coverage (88.9%, 40/45) on the proposed vulnerability taxonomy. Importantly, we attempted to incorporate test cases for the missing five vulnerability types from real-world industrial BNB projects; however, we were unable to locate any suitable examples. Our industry auditing experts confirm that the lack of these vulnerability types in our benchmark is due to their rare occurrence in practical projects. This observation further underscores the benchmark's alignment with the contemporary landscape of smart contract vulnerabilities. Meanwhile, the distribution of vulnerability samples presented in Table 2 reveals notable skew, with certain types represented by a significantly larger number of samples. This was primarily attributed to the inclusion of the SolidiFI Benchmark [25]. Despite the observed skew, the decision to include the SolidiFI Benchmark was driven by the lack of available alternatives. This made SolidiFI Benchmark an essential choice for our study, ensuring a broad and valuable dataset for our analysis . To the best of our knowledge, it is **the largest smart contract vulnerability benchmark in Solidity (ground truth) at the function level.**

3.4 Research Questions

Based on the selected tools, proposed taxonomy, and our benchmark, we aim to answer the following research questions (RQs):

3.4.1 RQ1: Coverage Analysis. To what extent do existing SAST tools support different vulnerability types?

This RQ first explores the actual coverage of these generalized SAST tools across the vulnerability types defined in our taxonomy (§ 3.2). We assess the tools' detection capabilities to understand their comprehensiveness in identifying a wide spectrum of vulnerabilities. To this end, we leverage the previously established mapping of vulnerability identifiers to our taxonomy in § 3.3.2, aiming to quantify the effectiveness of these tools in safeguarding smart contracts against diverse security threats. Additionally, we provide the number of SWC entries to which the vulnerability identifiers can be mapped (i.e., # SWC in Table 2). To quantitatively evaluate the scope of vulnerabilities supported by each tool against our taxonomy, we define Coverage (= $\frac{\text{\# Supported vulnerability types}}{\text{\# Vulnerability types in our taxonomy}}$) to measure *the proportion of vulnerability types within our taxonomy that a tool claim to identify.*

3.4.2 RQ2: Effectiveness Analysis. How effective are these SAST tools in detecting vulnerabilities on our benchmark?

This RQ is addressed from two angles: *1)* The capability of current tools to effectively analyze our benchmark, and *2)* the number of vulnerabilities identified within our benchmark by these tools. For the initial aspect, we conducted a comprehensive review of the outcomes of the analyses, categorizing them into three distinct groups based on their performance: Successful scans, Scans that failed due to timeouts, and Scans that failed due to compilation issues. Regarding the second aspect, we enumerated the vulnerabilities detected across various types within our taxonomy. Subsequently, we calculated Recall ($\frac{TP}{TP+FN}$), Precision ($\frac{TP}{TP+FP}$), and F1-score ($\frac{2 \times Recall \times Precision}{Recall+Precision}$), to evaluate the effectiveness of these tools at function level.

All experiments in this study were performed on a server equipped with 80 vCPUs (Intel® Xeon® Gold 6248 CPU @ 2.50 GHz ×2) and 188G of RAMs, which uses GNU/Linux Ubuntu 18.04 (64-bit) as the host operating system. To automate the execution of these tools, we extended the SmartBugs [54] framework and enabled all detectors capable of identifying security vulnerabilities, as well as leveraging the recommended runtime parameters in [44], in which they proposed a unified standard to eliminate the bias in the assessment process.

3.4.3 RQ3: Consistency Analysis. This research question focuses on two consistency analyses: *1)* Are the detection results consistent among these tools in terms of the detected vulnerability categories? *2)* How effective are these SAST tools when combining their detecting results?

Building upon the empirical findings from RQ2, this research question aims to dissect the disparities among tools in their vulnerability detection. By meticulously analyzing both the commonalities and unique attributes of these tools' detection capabilities, we aim to shed light on their distinct strengths and potential shortcomings. This analysis focuses on: ① How these tools vary in detection rates across vulnerability categories. ② The potential boost in detection when combining tools, leveraging their individual strengths.

3.4.4 RQ4: Efficiency Analysis. How efficient are these SAST tools to perform an analysis?

To thoroughly assess the efficiency of various SAST tools on real-world smart contracts, we analyze all 8,981 contract files collected in § 3.3.1. For a robust and reliable evaluation under diverse conditions, we consider the efficiency based on all 8,981 contract files regardless of the success or failure of the analysis. To ensure robustness and consider potential infrastructure variability, we performed each performance measurement three times for each tool. The reported results represent the average of these trials.

4 COMPARISON AND EVALUATION

4.1 RQ1: Coverage Analysis

As depicted in Table 2, the vulnerability coverage varies among tools, despite they are generalized-focused tools. A noteworthy observation is the significant discrepancy in coverage between the commercial tool CSA, which supports nearly all types except for "Arbitrary Function Jump via Inline Assembly" (achieving 97.78%, 44/45), and the open-source tools. Among these, Slither stands out with the second-highest coverage at 95.56% (43/45), followed by Securify2 at 82.22% (37/45). This variation, particularly with Oyente showing lower coverage, highlights the impact of tool maturity on the breadth of supported vulnerability types, with older tools like Oyente lagging in updates for newer vulnerabilities. Meanwhile, it further illuminates areas requiring attention from tool developers. Specifically, the *Storage & Memory* and *Cryptographic* categories present lower coverage across the board, indicating critical directions for future enhancements. Interestingly,
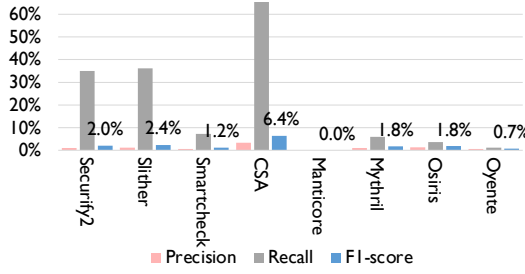
Fig. 3. Overall effectiveness of each tool on our benchmark.

Slither exhibits comprehensive coverage in these categories, positioning it as a benchmark for vulnerability detection in these domains.

> **Finding 1:** CSA, Slither, and Securify2 demonstrate the highest coverage across multiple categories. Slither excels in the *Reentrancy* and *Storage & Memory* categories, while CSA stands out in all categories except for *Storage & Memory*.

Notably, Slither ranks second in coverage yet misses two prevalent vulnerability types. Specifically, ① in the *Arithmetic* category, Slither fails to support *Integer Overflow/Underflow*, which are frequently identified in smart contracts [59]. While the Solidity compiler has built-in checks for these issues [61] from version `v0.8.0` [61], the inclusion of the unchecked keyword allows developers to bypass these protections for the sake of gas optimization. This bypass reintroduces potential overflow/underflow risks that the compiler's checks aimed to eliminate. It shows a crucial gap in Slither's analysis capabilities. Users should be aware of this when seeking overflow/underflow analysis, especially where compiler protections might be bypassed. ② Moreover, Slither lacks rules for detecting Transaction Order Dependence (TOD) vulnerabilities. These vulnerabilities are a common concern [64, 70, 75] and are supported by all other tools except for Manticore. The absence of TOD support in them (both developed by Trail of Bits) suggests a potential oversight and a critical area for improvement.

Other tools, such as Mythril, Manticore, and Osiris, display selective coverage, prioritizing specific vulnerability categories. For instance, Mythril concentrates on *Access Control*. Meanwhile, our analysis of tool support revealed that Manticore is equipped with three specific vulnerability identifiers for "Unsafe `Delegatecall`" within the *Access Control* category. This might suggest a targeted emphasis by Manticore on detecting this particular type of vulnerability. Moreover, despite having a smaller set of vulnerability identifiers, Osiris prioritizes the *Arithmetic* category with five out of its nine vulnerability identifiers targeting it. It covers four distinct vulnerability types within this category, demonstrating its granular focus despite limited overall coverage. However, Oyente and Osiris cover only four categories, potentially limiting their ability to detect a variety of vulnerabilities across categories.

> **Finding 2:** Other tools, such as Mythril, Manticore, and Osiris, show selective coverage, focusing on specific vulnerability categories. While such focused coverage can lead to detailed vulnerability identification in those categories, it also results in limited overall coverage.

4.2 RQ2: Effectiveness Analysis

As shown in Table 4, we first investigated the analysis status due to numerous cases where tools failed to analyze contracts or generate reports. We observed that existing tools primarily analyze

Table 4. Analysis status overview for each SAST tool. # Success and # Failure indicate the number of contracts where tools scan successfully and unsuccessfully, respectively. # Timeout indicates the number of contracts where the tool failed due to exceeding a time limit. # Compilation denotes the number of contracts that could not be scanned due to compilation errors.

Tool	# Success	# Failure		
		# Timeout	# Compilation	# Total
Securify2	247 (31.35%)	38	**503**	541
Slither	767 (97.34%)	2	19	21
SmartCheck	**788 (100.00%)**	0	0	0
CSA	772 (97.97%)	0	16	16
Manticore	112 (14.21%)	**626**	50	**676**
Mythril	590 (74.87%)	193	0	193
Osiris	451 (57.23%)	35	302	337
Oyente	499 (63.32%)	57	232	289

smart contracts by first compiling them, except for SmartCheck. However, this approach often leads to compilation issues, causing the analyzers to fail in their analysis. For instance, Securify2 relies on the compiler version of contracts under examination to generate its Abstract Syntax Tree (AST), but it is limited to compiler versions between v0.5.* and v0.6.*. As a result, Securify2 fails to analyze contracts developed using other compiler versions. Oyente faces challenges when analyzing contracts due to its limited support for compiler versions up to v0.4.19. Consequently, Oyente failed to analyze 232 (29.44%) contracts that were developed using higher solc versions with different AST structures [74]. Osiris, built on the foundation of Oyente, encountered a "compilation failed" error on 38.32% (302) contracts as it only supports the Solidity compiler v0.4.21. This limitation causes Osiris to struggle with contracts written in Solidity versions beyond v0.4.21, triggering compilation failures and preventing further vulnerability detection, rendering Osiris less efficient compared to other analyzers. Interestingly, Manticore faced difficulties in analyzing 626 out of 788 contract files. More specifically, it encountered a timeout in 79.44% (626/788) contracts and demanded substantial memory resources in our experimental system. Additionally, Manticore failed to analyze 50 contracts and generated exception errors due to compilation issues. To further investigate this issue, we focused on the SolidiFI Benchmark, where Manticore failed to analyze all the 300 contracts: 40 due to compilation issues and 260 due to timeout. For the 260 contracts that encountered timeouts, we removed the timeout constraint and reran Manticore to evaluate its detection capacity. However, we observed that over three weeks, Manticore successfully analyzed only 36 out of 260 contracts without yielding new findings. This underscores the practicality of our one-hour limit by demonstrating its sufficiency even for the most demanding analyses. Additionally, upon further exploration, we found that 44.25% (50/113) BNB projects encountered compilation failure when using Securify2, Manticore, and Oyente. We further investigated that tools except for CSA, Slither, and SmartCheck only support single-file analysis, resulting in them having no capacity to scan multiple files under a project and causing compilation failure, especially when these projects involve importing other files.

Finding 3: Some analyzers such as Oyente, Osiris, and Securify2 run deeply tied to the Solidity compiler version, which can cause analyzing failure when trying to scan smart contracts written in different versions.

As illustrated in Figure 3, the tools exhibit a wide range of effectiveness in vulnerability detection, with many performing below expectations. CSA stands out in terms of F1-score, with its metrics

revealing a Recall of 72.00%, Precision of 3.35%, and an F1-score of 6.40%. Slither follows with an F1-score of 2.38%, Recall of 36.18%, and Precision of 1.23%. Other tools, such as Manticore, show disappointing results, failing to detect any vulnerabilities in our benchmark, resulting in an F1-score of 0% due to numerous analysis failures. Interestingly, static analysis (SA) tools generally achieve better Recall, whereas symbolic execution (SE) tools tend to offer superior Precision.

Regarding SA tools, Intermediate Representation (IR) has become a cornerstone to translate high-level source code to actionable, low-level insights. All four SA tools studied (i.e., Securify2, Slither, SmartCheck, and CSA) employ IR, yet their quality varies. Both Slither and Securify2 harness IR by compiling the code, allowing them to conduct control flow graph (CFG) and data flow graph (DFG) analyses with enhanced precision [48, 51]. This meticulous compilation-based approach provides a robust depiction of code behavior during execution. In contrast, SmartCheck adopts ANTLR [41] to directly transform Solidity code into an XML-based IR [66]. This non-compilation approach enhances its analysis success rate, as it sidesteps potential compilation challenges mentioned before. However, the trade-off is its abstracted IR, which may lack the granularity to capture all code intricacies. This is underscored by SmartCheck's performance on the SolidiFI Benchmark: while it detects an average of $1,490$ vulnerabilities per contract, its precision stands at a mere 0.65%. This highlights the balance between the detailed methods of Slither and Securify2 versus the more generalized but potentially less precise approach of SmartCheck.

Generally, these SA tools struggle with high false positives, resulting in an average F1-score below 10%. An intriguing relationship emerges between false negatives and false positives: The four SA tools (Securify2, Slither, SmartCheck, and CSA) focused on source code and attained elevated Recall, which is attributed to their inherent technical constraints, which prioritize achieving a *sound* detection by compromising the *completeness* [5]. However, it results in a trade-off, manifesting in diminished Precision and an increased number of false positives.

Take Slither as an example, despite utilizing advanced detection techniques involving data-flow and control-flow analysis on the `SlithIR`, it still suffers from unexpected false positives. A notable instance is its detection logic for "Weak PRNG (Pseudorandom Number Generator)", which refers to compromised PRNG due to predictable or manipulable sources like using `blockhash`, with the primary concern being the security risk from predictable outcomes. However, the detector implementation within Slither [53] utilizes Static Single Assignment (SSA) form to identify potentially risky uses of the modulus operator when its left operand depends on `block.timestamp`, `now`, or `blockhash`. Despite this sophisticated approach, it frequently misidentifies benign code as vulnerable (resulting in false positives) in practice. For instance, as shown in Listing 1, the seven instances reported as "Weak PRNG" vulnerabilities in the PESA Token [7] were all false positives (L2, L5, L8, L14, L17, L20, and L23). However, the code demonstrates benign applications of the modulus operator for various logical operations, such as determining leap years (L1-L12) or extracting time components from a timestamp (L13-L24). This false positive originates from the simplistic approach of the detection heuristic for this vulnerability pattern, which lacks a nuanced understanding necessary to differentiate between the use of the modulus operator in random number generation and its benign use in straightforward arithmetic operations.

In contrast, SE tools including Mythril, Oyente, and Osiris, which analyze bytecode, tend to be more precise when compared with these SA tools. The enhanced precision arises from the fact that SE simulates specific code execution paths. However, while this offers a more meticulous analysis, it can sometimes overlook vulnerabilities outside the simulated paths, particularly when utilizing Depth-First Search (DFS) within constrained timeframes.

```
1   function isLeapYear(uint16 year) public pure returns (bool) {
2       if (year % 4 != 0) { // FP-1
3               return false;
4       }
5       if (year % 100 != 0) { // FP-2
6               return true;
7       }
8       if (year % 400 != 0) { // FP-3
9               return false;
10      }
11      return true;
12  }
13  function getHour(uint timestamp) public pure returns (uint8) {
14          return uint8((timestamp / 60 / 60) % 24); // FP-4
15  }
16  function getMinute(uint timestamp) public pure returns (uint8) {
17          return uint8((timestamp / 60) % 60); // FP-5
18  }
19  function getSecond(uint timestamp) public pure returns (uint8) {
20          return uint8(timestamp % 60); // FP-6
21  }
22  function getWeekday(uint timestamp) public pure returns (uint8) {
23          return uint8((timestamp / DAY_IN_SECONDS + 4) % 7); // FP-7
24  }
```

Listing 1. Examples of benign modulus operator usage incorrectly flagged as Weak PRNG vulnerabilities by Slither in the PESA Token [7].

> **Finding 4:** SAST tool effectiveness varies notably. CSA and Slither lead with F1-scores of 6.40% and 2.38%, while Oyente and Manticore underperform. The quality of IR is crucial: detailed IRs enhance accuracy, but SmartCheck's approach leads to high false positives. SA (using source code analysis) boosts Recall, while SE (using bytecode analysis) prioritizes Precision.

4.3 RQ3: Consistency Analysis

Detection consistency across vulnerability categories. Generally, these tools exhibited weak detection capabilities across all categories, which can be inferred from the results in RQ1. Interestingly, we still observed a significant inconsistency in the detection capabilities of SAST tools across various vulnerability categories, as displayed in Table 5. For instance, these tools exhibited more effective detection rates for *Access Control* (30.31%) and *Reentrancy* (28.03%) vulnerabilities when compared to other categories.

In contrast, the tools demonstrated weaker effectiveness in *Storage & Memory* (1.88%), *Arithmetic* (7.05%), and *Cryptographic* (12.50%) categories. Notably, while all tools support *Arithmetic* vulnerabilities, they detected no more than 40% of them. The observed inconsistency also sheds light on the distinct methodologies and detection focuses of different SAST tools. For example, while most tools detected an average of 16.23% *Denial of Service* vulnerabilities, Securify2 single-handedly detected 64% of the 1,389 vulnerabilities. This difference suggests Securify2's unique approach. It combines both "compliance" and "violation" patterns for formal verification instead of relying solely on known sensitive signatures or heuristics. A prime example is its detection pattern for "`transfer()` and `send()` with Hardcoded Gas Amount". As depicted in Listing 2, Securify2 first identifies contexts where external calls are made (L2). For compliance, it verifies if the ether amount associated with a call is zero (L4-L7) or if the external call explicitly depends on the transaction's sender (i.e., `msg.sender`) (L10-L15). Finally, violations are flagged when these compliant conditions are not met, signaling potential unrestricted ether flows (L18-L22). However, the inherent drawback

```
1  // Identifying relevant contexts via external calls
2  applicableInContext(callCtx) :- externalCall(call), ctxProvider.elementInContext(callCtx,
       call, _).
3  // Compliance check: Ether amount associated with a call is zero
4  compliantInContext(callCtx, "") :-
5      applicableInContext(callCtx), ctxProvider.elementInContext(callCtx, call, context),
6      callValue(call, amount),
7      valueOf([amount, context], "0").
8  // Compliance check: External call depends on the transaction's sender
9  compliantInContext(callCtx, "") :-
10     applicableInContext(callCtx),
11     someSender(senderCtx),
12     (
13         infoflow.instrMustDependOn(callCtx, senderCtx)
14     ).
15 // Violation rule: Non-compliant conditions, indicating potential unrestricted ether
        flows
16 violationInContext(callCtx, "") :-
17     applicableInContext(callCtx), ctxProvider.elementInContext(callCtx, call, context),
18     callValue(call, value),
19     !zeroValue([value, context]),
20     !maybeCompliantInContext(callCtx).
```

Listing 2. Detection logic for *unrestricted-ether-flow* within Securify2.

Table 5. The number of detected vulnerabilities in each category. For each vulnerability category: AC: *Access Control*, AR: *Arithmetic*, BM: *Block Manipulation*, CR: *Cryptographic*, DoS: *Denial of Services*, RE: *Reentrancy*, S&M: *Storage & Memory*.

Tool	AC	AR	BM	CR	DoS	RE	S&M
Securify2	909	0	1,602	0	883	236	0
Slither	1,306	10	1,123	1	69	1,228	3
SmartCheck	57	1	480	0	15	201	0
CSA	1,465	651	3,125	1	835	1,374	33
Manticore	0	0	0	0	0	0	0
Mythril	264	44	210	0	2	98	0
Osiris	1	326	4	0	0	47	0
Oyente	1	91	0	0	0	32	0
# Vulns in our benchmark	1,651	1,992	3,886	2	1,389	1,434	40

of Securify2 relying on hard-coded patterns, such as *mul-after-div.dl* [49], becomes evident when faced with intricate arithmetic issues in real-world scenarios, especially in DeFi apps where diverse factors intertwine, leading to potential miscalculations or undetected vulnerabilities.

Meanwhile, CSA and Slither led in detecting *Reentrancy* vulnerabilities (95.82% and 85.63%, respectively). We found that Slither employs a heuristic-based method to detect potential reentrancy vulnerabilities in smart contracts. By integrating 7 detectors for reentrancy detection [40, 52], it systematically traverses the CFG of each implemented function, searching for patterns that could indicate reentrancy, particularly scenarios where state variables are written after an external call. However, Slither still has its limitations in detecting this category, especially when analyzing contracts involving state variables. If the callee is a state variable controlled only by an administrator, Slither might mistakenly report a non-existent vulnerability (false positive). Conversely, if the callee is a state variable that can be influenced by anyone, Slither may overlook the vulnerability (false negative).

It is crucial for developers and researchers to recognize these strengths and shortcomings. This comprehension can help guide further advancements in SA tools, ensuring that they continue to evolve and offer robust defenses against potential smart contract vulnerabilities.

> **Finding 5:** There is a notable difference in detection rates across vulnerability categories. Interestingly, they identified more *Access Control* and *Reentrancy* vulnerabilities when compared to those belonging to *Storage & Memory*, *Arithmetic*, and *Cryptographic* categories. This inconsistency underscores their unique detection methodologies and the inherent challenges in improving the coverage for diverse vulnerability types.

The combination of tools. Since none of the single tools performs well on our benchmark and each tool has different focuses, we tried to analyze the effectiveness improvement by tool combination. ① *An inclusive combination strategy:* Here, we selected and combined the top-performing tools in terms of Recall: CSA (72.00%), Slither (36.18%), and Securify2 (34.92%) (see Figure 4). A vulnerability is thereby considered found if at least one tool from the respective group was able to detect it. Interestingly, our findings demonstrate that CSA and Securify2 effectively complement each other in vulnerability detection. Specifically, as shown in Figure 4 (b), they identified 91.5% (9,513/10,394) compared to the best-performing single tool, CSA (72.00%). However, this improvement comes at the cost of marking 36.77 percentage points more functions as potentially vulnerable (271,794 reported issues in total).

> **Finding 6:** Applying an *inclusive combination* strategy, where a vulnerability is considered found if detected by any tool, substantially raises Recall to 91.5%, albeit at the cost of decreasing Precision to 2.67%.

② *Majority voting strategy:* The low Precision of single tools motivated us to explore consensus-based approaches for potentially improving Precision. To this end, we adopted the "majority voting" [28] strategy, excluding Manticore due to its inability to detect any vulnerabilities, and thus considering the consensus among the remaining seven tools. This approach yielded a noteworthy increase in Precision to 36.10% (717/1,986), with a corresponding improvement in the F1-score to 11.58%. Compared to the single tool with the highest Precision, CSA (3.35%), this consensus-based approach significantly improved Precision. However, Recall was reduced to 6.90%, which underlines the trade-off between Precision and Recall and the inherently conservative nature of the majority voting strategy.
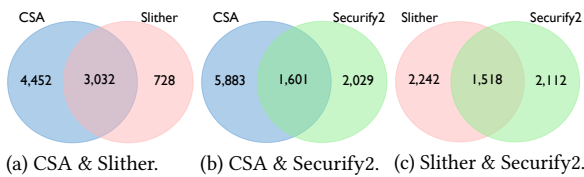
Table 6. Analysis duration of each tool.

Tool	Duration	
	Average (s)	Total (s)
Securify2	49.7	446,741.0
Slither	16.6	149,165.3
SmartCheck	112.7	1,012,242.0
CSA	17.5	157,081.3
Manticore	1,547.6	13,898,788.1
Mythril	579.0	5,199,823.6
Osiris	51.7	464,279.6
Oyente	13.2	118,301.1

(a) CSA & Slither.   (b) CSA & Securify2.   (c) Slither & Securify2.

Fig. 4. Combination of SAST tools.

> **Finding 7:** The *majority voting* among tools improves Precision to 36.10%, increases the F1-score to 11.58%, but decreases Recall to 6.90%. This approach reduces SAST tools' false alarms, emphasizing the precision-recall trade-off.

4.4   RQ4: Efficiency Analysis

As illustrated in Table 6, it presents the average and total execution time of each tool when analyzing the selected smart contracts. On average, it takes about 298.5 seconds for each tool to analyze a contract. Among SA tools, Slither and CSA complete their analyses within 16.6 seconds and 17.5 seconds per contract on average, respectively, while SmartCheck tends to be less efficient, taking approximately 112.7 seconds on average. Notably, Securify2 demands more memory resources during analysis than other tools, a factor that users with resource constraints should consider.

In contrast, SE tools require substantially more time to execute. Manticore and Mythril have average analysis durations of $1,547.6s$ and $579.0s$, respectively. Oyente appears to be the fastest tool, taking an average of just $13.2s$ to analyze a contract. However, this is primarily due to the numerous compilation failures it encountered $(8,355/8,981)$, as already shown in Table 4, which caused it to exit immediately when encountering such issues. Similarly, although Osiris is not as fast as Oyente, it is worth noting that it also had a significant number of compilation failures $(8,533/8,981)$, which resulted in its relatively faster analysis time of $51.7s$.

In general, SA tools are faster than SE tools due to state traversal challenges faced by the latter. SA tools such as Securify2 and Slither, are faster due to their use of IR and the efficient program analysis techniques they employ. For instance, Securify2 transforms source code into an IR using the MLton compiler, an optimizer for the Standard ML programming language [24]. Their faster execution time makes these tools more suitable for analyzing extensive contract sets or time-sensitive application scenarios. SE tools, such as Mythril, often have longer execution times due to their technical nature, which explores all possible execution paths, leading to vast state space. This comprehensive exploration, while time-consuming, allows SE tools to uncover intricate vulnerabilities and edge cases, providing a deeper understanding of potential issues in smart contracts.

> **Finding 8:** SA tools generally outpace SE tools in terms of execution speed. Slither and CSA stand out with their swift scanning speed. In contrast, SE tools like Manticore and Mythril take considerably longer due to their exhaustive detection nature. While SA tools offer speed, SE tools provide depth, uncovering complex vulnerabilities by exploring all possible execution paths.

5   DISCUSSION

5.1   Implications

5.1.1   For Tool Developers. Based on our findings, we suggest the following improvements for SAST tool developers: ① *Handle compilation issues more robustly.* Our analysis in § 4.1 shows that these tools generally failed to perform analysis due to compilation issues. Future research should focus on developing tools that can accommodate a wider range of compiler versions and handle compilation issues more robustly. This may involve incorporating techniques such as partial compilation or on-the-fly compiler version switching. By enhancing the compatibility of these tools with various compiler versions, the effectiveness and reliability of smart contract analysis can be significantly improved. ② *Improve Precision by refining detection implementation.* As shown in § 4.2, existing tools, including the commercial tool CSA, generally face a dilemma where they report numerous issues when analyzing smart contracts. As depicted in § 4.3, one potential cause could be the heavy reliance on pattern matching for detection. The coverage and accuracy of these predefined rules can greatly influence effectiveness. Developers are expected to refine the detection rules by capturing the exact semantics of existing vulnerabilities rather than merely using intuitive pattern-matching. For instance, when detecting *Reentrancy* vulnerabilities, tools are expected to capture the state changes rather than a simple search for call.value. ③ *Focus on the interoperability of tools.* The

findings in § 4.3 demonstrate that some tools perform better in certain vulnerability categories than others. Therefore, it is essential to consider the interoperability of different tools, allowing users to combine the strengths of multiple tools for a more comprehensive security analysis. This can be achieved by adopting standardized input and output formats and developing frameworks that facilitate the integration of multiple tools. ④ *Improve the efficiency of tools.* Our study shows that some tools, such as Manticore analyzed contracts much slower than expected (1,547.6 seconds per contract), which can negatively impact their practical effectiveness. In particular, Manticore does not support a customized `max recursion depth` setting for users when compared to the other SE tools. To address this issue, developers should focus on optimizing tool performance by implementing more efficient algorithms, providing better resource management, and offering user-configurable settings for fine-tuning tool behavior based on specific requirements.

5.1.2 For Researchers. We encourage future research in smart contract security to: ① *Propose a more comprehensive and up-to-date taxonomy and develop a diverse benchmark.* Taxonomies facilitate benchmark construction and objective evaluation in the context of smart contracts' development. As smart contract technologies evolve, new vulnerability types may emerge, necessitating a continuously updated, unified, and comprehensive taxonomy. Simultaneously, we encourage the development of a comprehensive and diverse benchmark for objectively evaluating SAST tools in line with the evolving taxonomy. ② *Call for a flexible IR design in SAST for smart contracts.* As discussed in § 4.2, Slither and Securify2's compilation-based IR approach provides detailed analyses, while SmartCheck's direct transformation method, although resolving compilation challenges, might lack depth. IR profoundly affects the Precision and success rate of SAST tools, emphasizing the need for a balance between granularity and analysis success rate. To improve these tools, developers could explore hybrid IR strategies that combine the strengths of both compilation-based (IR) and direct transformation techniques. Continuous refinement based on feedback and benchmark performance, coupled with collaborative research, could pave the way for more robust IR methodologies that capture code intricacies without compromising the analysis success rate. ③ *Better balance Recall and Precision in SAST tools for smart contracts.* Tools using SA generally achieve higher Recall but suffer from false positives, while SE tools exhibit better Precision but are limited by higher true positives due to their inherent technological limitations. Identifying effective strategies for balancing Recall and Precision presents a challenge for future research in the field of smart contract security analysis.

5.1.3 For Practitioners. For practitioners working with smart contract SAST tools, we offer the following practical guidance: ① *Select tools carefully by considering their strengths and limitations.* Be aware that some tools may have limitations due to compilation issues and may not support the latest compiler versions. ② *The strategic application of multiple SAST tools, based on their unique strengths, can cater to specific needs in vulnerability detection.* As shown in § 4.3, tools demonstrate diverse strengths, with Securify2 excelling in detecting *Denial of Service* vulnerabilities and Slither displaying proficiency in *Reentrancy* issues. Importantly, the choice of combination strategy can greatly impact the Precision-Recall balance.

5.2 Threats to Validity

5.2.1 External Validity. ① The first threat concerns the generalizability of our evaluation results, specifically whether our findings can be applied to other SAST tools and datasets for Solidity smart contracts. We mitigated it by collecting existing benchmarks plus 113 real-world BNB projects, which contain validated vulnerabilities mapped and grouped into 45 unique vulnerability types in our taxonomy. Furthermore, we selected eight state-of-the-art tools (seven open-source and one commercial) that employ various techniques and are frequently used and/or evaluated in recent

top-tier publications and prominent vendors. ② Another threat comes from the four collected benchmarks. As discussed in § 3.3.1, we collected benchmarks with "labeled vulnerability types" and labeled vulnerable functions based on their claimed types. During this process, we identified and labeled unclaimed vulnerabilities, such as 1,611 new vulnerabilities in the SolidiFI Benchmark compared to their claimed 7,995. To address this, we thoroughly labeled additional vulnerabilities for a comprehensive evaluation of the tools' Precision. ③ The last threat concerns the skewed distribution within our benchmark (§ 3.3.2). This skew could potentially bias the overall effectiveness results. To mitigate it, we provide a detailed breakdown of the detection results on each vulnerability type in Table 2, aiming to offer a clearer insight into the tools' effectiveness.

5.2.2 Internal Validity. ① A possible threat arises from mistakes in labeling and mapping for ground truth and detection rules of the selected tools. To mitigate this, we collaborated with our industry partner and engaged three security experts to label the ground truth at the function level, a process that took a total of 7.5 person-months. Furthermore, three security auditing experts independently mapped the ground truth and detection rules, discussing any conflicts in mapping results until reaching an agreement, which required an additional 3.5 person-months. ② Another threat involves the runtime parameters used during the execution of the tools, particularly the one-hour timeout setting. We adopted this timeout based on recommendations in [44]. Notably, our in-depth analysis on Manticore in § 4.2 highlighted its practicality: despite facing numerous timeouts, a constraint-free rerun of Manticore on the SolidiFI benchmark led to analyzing only 36 out of 260 contracts, without any discoveries. This outcome, coupled with the analysis in § 4.4 and Table 6, affirms the one-hour limit's adequacy for most contracts.

6 CONCLUSION AND FUTURE WORK

This paper presents a comprehensive evaluation of eight SAST tools for Solidity smart contracts including coverage, effectiveness, consistency, and efficiency. We identify notable gaps in rule coverage, with tools struggling to comprehensively support the evolving range of Solidity vulnerabilities. Critical issues include high false positives in static analysis/source-code-based tools and symbolic execution/bytecode-based tools' limited path coverage. We highlight the urgent need for tool advancements to address compiler limitations and introduce more effective strategies, such as partial compilation or version adaptability. The varied effectiveness across vulnerability types suggests a potential for refining specialized tools. Emphasizing the need for improved detection methods, we suggest exploring hybrid analysis for complex vulnerabilities like reentrancy. Future research should consider improving the detection implementation by capturing the exact vulnerability patterns and exploring sophisticated tool combination strategies, and efficiency enhancements (especially for Manticore) to bolster SAST tool performance. Our insights guide tool enhancement efforts, contributing to the security of the blockchain ecosystem.

REFERENCES

[1] Sefa Akca, Chao Peng, and Ajitha Rajan. 2021. Testing Smart Contracts: Which Technique Performs Best?. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (Bari, Italy) *(ESEM '21)*. Association for Computing Machinery, New York, NY, USA, Article 21, 11 pages. https://doi.org/10.1145/3475716.3475779

[2] Bowen Alpern, Mark N Wegman, and F Kenneth Zadeck. 1988. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1–11.

[3] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*. 1–15.

[4] Clark Barrett and Cesare Tinelli. 2018. *Satisfiability Modulo Theories*. Springer International Publishing, Cham, 305–343. https://doi.org/10.1007/978-3-319-10575-8_11

[5] George S Boolos, John P Burgess, and Richard C Jeffrey. 2002. *Computability and logic*. Cambridge university press.

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

[7] BSCScan. 2024. Pesabase: PESA Token | Address 0x4adc604A0261E3D340745533964FFf6bB130f3c3 | BscScan. https://bscscan.com/address/0x4adc604A0261E3D340745533964FFf6bB130f3c3#code. (Accessed on 02/24/2024).

[8] BNB Beacon Chain. 2023. Binance (BNB) Blockchain Explorer. https://bscscan.com/. (Accessed on 05/02/2024).

[9] ChainSecurity. 2018. eth-sri/securify: [DEPRECATED] Security Scanner for Ethereum Smart Contracts. https://github.com/eth-sri/securify. (Accessed on 05/02/2024).

[10] ChainSecurity. 2020. eth-sri/securify2: Securify v2.0. https://github.com/eth-sri/securify2. (Accessed on 05/02/2024).

[11] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Ben Livshits. 2023. Smart Contract and DeFi Security: Insights from Tool Evaluations and Practitioner Surveys. *arXiv preprint arXiv:2304.02981* (2023).

[12] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Benjamin Livshits. 2023. Smart Contract and DeFi Security Tools: Do They Meet the Needs of Practitioners?. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 705–717.

[13] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2022. Defining Smart Contract Defects on Ethereum. *IEEE Transactions on Software Engineering* 48, 1 (2022), 327–345. https://doi.org/10.1109/TSE.2020.2989002

[14] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2022. Defining Smart Contract Defects on Ethereum. *IEEE Transactions on Software Engineering* 48, 1 (2022), 327–345. https://doi.org/10.1109/TSE.2020.2989002

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

[16] ConsenSys. 2018. ConsenSys/mythril: Security analysis tool for EVM bytecode. https://github.com/ConsenSys/mythril. (Accessed on 05/02/2024).

[17] Contractlogix. 2022. Smart Contract Security in 2023: A Simple Checklist. https://www.contractlogix.com/contract-management/smart-contract-security/. (Accessed on 05/02/2024).

[18] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1989. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 25–35.

[19] Ardit Dika and Mariusz Nowostawski. 2018. Security Vulnerabilities in Ethereum Smart Contracts. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. 955–962. https://doi.org/10.1109/Cybermatics_2018.2018.00182

[20] 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

[21] Ethlint. 2024. duaraghav8/Ethlint. https://github.com/duaraghav8/Ethlint. (Accessed on 05/02/2024).

[22] Yuzhou Fang, Daoyuan Wu, Xiao Yi, Shuai Wang, Yufan Chen, Mengjie Chen, Yang Liu, and Lingxiao Jiang. 2023. Beyond "Protected" and "Private": An Empirical Security Analysis of Custom Function Modifiers in Smart Contracts. (2023).

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

[24] Matthew Fluet. 2021. MLton. http://www.mlton.org/. (Accessed on 10/09/2023).

[25] 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* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 415–427. https://doi.org/10.1145/3395363.3397385

[26] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. [n. d.]. AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities. ([n. d.]).

[27] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2022. ETainter: Detecting Gas-Related Vulnerabilities in Smart Contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 728–739. https://doi.org/10.1145/3533767.3534378

[28] Jerónimo Hernández-González, Daniel Rodriguez, Inaki Inza, Rachel Harrison, and Jose A Lozano. 2018. Learning to classify software defects from crowds: a novel approach. *Applied Soft Computing* 62 (2018), 579–591.

[29] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 259–269.

[30] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts.. In *Ndss*. 1–12.

[31] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association. https://publications.cispa.saarland/2612/

[32] Zeqin Liao, Zibin Zheng, Xiao Chen, and Yuhong Nan. 2022. SmartDagger: A Bytecode-Based Static Analysis Approach for Detecting Cross-Contract Vulnerability. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 752–764. https://doi.org/10.1145/3533767.3534222

[33] 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*. 65–68.

[34] Ye Liu, Yi Li, Shang-Wei Lin, and Rong Zhao. 2020. Towards automated verification of smart contract fairness. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 666–677.

[35] António Pedro Cruz Monteiro. 2019. A study of static analysis tools for ethereum smart contracts. (2019).

[36] 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

[37] MythX. 2023. MythX: Smart contract security service for Ethereum. https://mythx.io/. (Accessed on 05/02/2024).

[38] NCC Group. 2016. DASP - TOP 10. https://dasp.co/. (Accessed on 05/02/2024).

[39] Trail of Bits. 2023. Trail of Bits. https://www.trailofbits.com/. (Accessed on 05/02/2024).

[40] Website of Our Study. 2024. RQ2. https://sites.google.com/view/sc-sast-study-fse2024/rq2. (Accessed on 01/03/2024).

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

[42] 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)*. IEEE, 1661–1677.

[43] Heidelinde Rameder. 2021. Systematic review of ethereum smart contract security vulnerabilities, analysis methods and tools. (2021).

[44] 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

[45] Microsoft Research. 2023. The Z3 Theorem Prover. https://github.com/Z3Prover/z3. (Accessed on 05/02/2024).

[46] 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).

[47] BK RWZ88 and MN Rosen. 1988. Wegman, and FK Zadeck. Global value numbers and redundant computations. In *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages*.

[48] Securify2. 2020. securify2/securify/ir. https://github.com/eth-sri/securify2/tree/master/securify/ir. (Accessed on 10/09/2023).

[49] Securify2. 2020. securify2/securify/staticanalysis/souffle_analysis/patterns/mul-after-div.dl. https://github.com/eth-sri/securify2/blob/def1e30ba9198828d048fbba5fbb6cd27f7e1b04/securify/staticanalysis/souffle_analysis/patterns/

mul-after-div.dl. (Accessed on 10/09/2023).

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

[51] Slither. 2023. slither/slither/slithir. https://github.com/crytic/slither/tree/e5f2a86f0906fd62c6c4eccb9dbfa5ab30671a78/slither/slithir. (Accessed on 10/09/2023).

[52] Slither. 2024. crytic/slither. https://github.com/crytic/slither/tree/e3dcf1ecd3e9de60da046de471c5663ab637993a/slither/detectors/reentrancy. (Accessed on 01/03/2024).

[53] Slither. 2024. slither/slither/detectors/operations/bad_prng.py at efed98327a7553badfd1c56720136637885b9207 · crytic/slither · GitHub. https://github.com/crytic/slither/blob/efed98327a7553badfd1c56720136637885b9207/slither/detectors/operations/bad_prng.py. (Accessed on 25/02/2024).

[54] Smartbugs. 2020. SmartBugs: A Framework to Analyze Ethereum Smart Contracts. https://github.com/smartbugs/smartbugs. (Accessed on 05/02/2024).

[55] SmartBugs. 2020. smartbugs-wild. https://github.com/smartbugs/smartbugs-wild. (Accessed on 05/02/2024).

[56] SmartBugs. 2022. smartbugs-curated. https://github.com/smartbugs/smartbugs-curated. (Accessed on 31/03/2023).

[57] SmartDec. 2023. SmartDec. https://smartdec.net/. (Accessed on 05/02/2024).

[58] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. {SmarTest}: Effectively hunting vulnerable transaction sequences in smart contracts through language {Model-Guided} symbolic execution. In *30th USENIX Security Symposium (USENIX Security 21)*. 1361–1378.

[59] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. 2020. VeriSmart: A highly precise safety verifier for Ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1678–1694.

[60] Solhint. 2024. protofire/solhint. https://github.com/protofire/solhint. (Accessed on 05/02/2024).

[61] Solidity. 2024. Solidity v0.8.0 Breaking Changes — Solidity 0.8.25 documentation. https://docs.soliditylang.org/en/latest/080-breaking-changes.html. (Accessed on 29/02/2024).

[62] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. 2021. SmartPulse: automated checking of temporal properties in smart contracts. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 555–571.

[63] Kairan Sun, Zhengzi Xu, Chengwei Liu, Kaixuan Li, and Yang Liu. 2023. Demystifying the Composition and Code Reuse in Solidity Smart Contracts. In *Proceedings of the 31th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3611643.3616270

[64] SWC-114. 2024. SWC-114 - Smart Contract Weakness Classification (SWC). https://swcregistry.io/docs/SWC-114/. (Accessed on 29/02/2024).

[65] SWC Registry. 2018. Overview · Smart Contract Weakness Classification and Test Cases. https://swcregistry.io/. (Accessed on 05/02/2024).

[66] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain* (Gothenburg, Sweden) *(WETSEB '18)*. Association for Computing Machinery, New York, NY, USA, 9–16. https://doi.org/10.1145/3194113.3194115

[67] Christof Ferreira Torres, Julian Schütte, 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.

[68] Trail of Bits. 2017. crytic/not-so-smart-contracts: Examples of Solidity security issues. https://github.com/crytic/not-so-smart-contracts. (Accessed on 05/02/2024).

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

[70] Shuai Wang, Chengyu Zhang, and Zhendong Su. 2019. Detecting nondeterministic payment bugs in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[71] Website of Our Study. 2023. Static Application Security Testing (SAST) Tools for Smart Contracts: How Far Are We? https://sites.google.com/view/sc-sast-study-fse2024/home. (Accessed on 05/02/2024).

[72] Website of Our Study. 2023. Tool Selection. https://sites.google.com/view/sc-sast-study-fse2024/tool-selection. (Accessed on 05/02/2024).

[73] 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

[74] yxliang01. 2020. oyente/oyente.py at enzymefinance/oyente GitHub. https://github.com/enzymefinance/oyente/blob/69dc0a905d37ae27e9055ccae930e30752b398fb/oyente/oyente.py#L63. (Accessed on 05/02/2024).

[75] Wuqi Zhang, Lili Wei, Shing-Chi Cheung, Yepang Liu, Shuqing Li, Lu Liu, and Michael R Lyu. 2022. Front-Running Attack Benchmark Construction and Vulnerability Detection Technique Evaluation. *arXiv preprint arXiv:2212.12110* (2022).

[76] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying Exploitable Bugs in Smart Contracts. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 615–627. https://doi.org/10.1109/ICSE48619.2023.00061

[77] 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.