Are Static Analysis Tools Still Working during the Evolution of Smart Contracts? A Comprehensive Empirical Study

CUIFENG GAO, AO CHEN, and CHENGZE WU, School of Computer Science and Technology, University of Science and Technology of China, China and Suzhou Institute for Advanced Research, University of Science and Technology of China, China

WENZHANG YANG, Institute of AI for Industries, Chinese Academy of Sciences, China

JIAMING YE, Southwest Jiaotong University, China

YINXING XUE*, Institute of AI for Industries, Chinese Academy of Sciences, China

Since the inception of Ethereum, numerous static analysis tools for Solidity-based smart contracts have been developed. During this time, Ethereum has rapidly evolved, undergoing 19 hard forks and five major breaking changes in the Solidity language, which can directly impact the capabilities of these tools. Existing research is often constrained to specific versions and tools, without offering a systematic and comprehensive evaluation of how these evolutions affect static analysis tools. However, assessing the full impact of these upgrades on the tools is far from straightforward.

To address this, we present a large-scale comprehensive empirical evaluation that employs 22 state-of-the-art static analysis tools and 251,340 unique contracts verified on Etherscan to systematically evaluate the compatibility, adaptability and consistency of these tools throughout the evolution of smart contracts. First, we conduct a pilot study on a representative sample subset of the full dataset to evaluate the compatibility of the tools by examining their execution success rates across different versions. Our findings show that only 10 out of 22 tools are fully compatible with all versions. Based on these findings, we perform a large-scale evaluation on the full dataset to explore the adaptability of these 10 tools across different versions. Our results reveal that, out of 10 tools, nine do not maintain stable detection rates, five exhibit significant TP rate fluctuations, and seven show notable degradation in effectiveness. Finally, we further evaluate the consistency of these 10 tools by comparing their detection results on the full dataset, showing that consistency tends to decrease as the versions are upgraded. Additionally, we analyze the reason behind limitations of static tools and discuss the contract semantic drift over time, offering valuable insights for future research.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Smart Contracts, Evolution, Static Analysis Tools, Evaluation

*corresponding author

Authors' Contact Information: Cuifeng Gao, gcf20225162@mail.ustc.edu.cn; Ao Chen, ca2018@mail.ustc.edu.cn; Chengze Wu, wuchengze@mail.ustc.edu.cn, School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, China and Suzhou Institute for Advanced Research, University of Science and Technology of China, Suzhou, Jiangsu, China; Wenzhang Yang, wzhyang@iaii.ac.cn, Institute of AI for Industries, Chinese Academy of Sciences, Nanjing, Jiangsu, China; Jiaming Ye, yejiaming@swjtu.edu.cn, Southwest Jiaotong University, Chengdu, Sichuan, China; Yinxing Xue, yxxue@iaii.ac.cn, Institute of AI for Industries, Chinese Academy of Sciences, Nanjing, Jiangsu, China.

Manuscript submitted to ACM

ACM Reference Format:
Cuifeng Gao, Ao Chen, Chengze Wu, Wenzhang Yang, Jiaming Ye, and Yinxing Xue. 2025. Are Static Analysis Tools Still Working during the Evolution of Smart Contracts? A Comprehensive Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (December 2025), 30 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Since going live in 2015, Ethereum has undergone nearly a decade of rapid evolution, with 19 major changes introduced through hard forks. Meanwhile, Solidity, the most widely used smart contract development language on Ethereum, has also witnessed frequent upgrades, with five significant breaking changes from minor version 0.4.x to 0.8.x. However, the evolution of Ethereum and Solidity has brought changes to the syntax and semantics of smart contracts, both at the opcode and source code levels, which impact the capabilities and effectiveness of static analysis tools.

In recent years, the evolution of Ethereum and Solidity has increasingly attracted the attention of researchers. However, a comprehensive and systematic evaluation of various versions and tools, along with in-depth insights, remains lacking. For example, as early as 2020, Hwang *et al.* [31] investigated vulnerabilities changes only between versions 0.4.x and 0.5.x, but did not assess the impact on security tools. Then, in 2023, Kado *et al.* [33] focused exclusively on the performance of SmartCheck in different versions, without considering other tools. Most recently, Di Angelo *et al.* [17] (2024) evaluated the evolution of bytecode-level tools using a bytecode dataset, which constrained the analysis of underlying causes. In summary, the studies mentioned above lack a systematic and comprehensive evaluation of the impact of version upgrades on static analysis tools.

In this work, we conduct a large-scale comprehensive empirical study to explore the impact of version upgrades on all available static analysis tools for smart contracts. Following a systematic literature review, we identify 246 smart contract security tools and then filter 22 static vulnerability detection tools among them with open source code and command-line interface. We leverage the well-known, largest publicly available source code dataset, Smart Contract Sanctuary - Ethereum (SCSE) [43]. After removing duplicates, this dataset contains 251,340 unique contracts across versions 0.4.x to 0.8.x. We perform a systematic triple analysis that focuses on the compatibility, adaptability, and consistency of these tools.

First, the compatibility refers to the ability of a tool to execute successfully across different versions of smart contracts. We conduct a pilot study to evaluate the compatibility of the tools by measuring the execution success rate of each tool. Considering the fixed performance of state-of-the-art tools, we randomly select samples from each minor version of Solidity using a 5% confidence interval and a 10% confidence level to effectively and efficiently test the tools. A tool is considered compatible with a version if it successfully executes at least 50% of the contracts without throwing any exceptions. Our findings indicate that only 45% (10 out of 22) of tools are compatible in all versions. We also identify key factors affecting compatibility, primarily resulting from changes in language specifications, compiler output, and the increasing complexity of contracts.

Second, the adaptability refers to the ability of a tool to report a reasonable number of vulnerabilities and maintain a stable detection accuracy across different versions. Based on the findings from the compatibility evaluation, we subsequently conduct an exploratory study on the full large-scale dataset to assess the adaptability of the 10 tools that are compatible with all versions. Given the absence of ground-truth labels in the dataset, we first evaluate tool adaptability by analyzing the trends in their detection rates for five high-risk vulnerabilities across different versions. We examine whether the detection rate trends of each tool aligned with those of other tools. Our results reveal that nine out of 10 tools exhibit anomalous trends in certain versions, indicating poor adaptability. Then, to gain further

insight into the actual detection capabilities of the tools, we independently sample detection results for each tool across versions and vulnerability types, manually validate their true positive (TP) rates, and find that half of the tools exhibit significant fluctuations. Additionally, to enable a more controlled evaluation, we construct a derived ground-truth dataset by aggregating the results of our manual validation across different tools. Using this dataset, we evaluate the effectiveness of the tools—including precision, recall, and F1 score—across different versions. The results show that seven tools exhibit notable degradation in effectiveness over time.

Third, the consistency refers to the similarity or overlap in the vulnerable contracts reported by the tools. For the ten tools that are compatible with all versions, we evaluate their consistency by analyzing the overlap in the detection results for the five high-risk vulnerabilities. We compute the average Jaccard Similarity Coefficient [32] for different tool combinations across various versions. Our findings indicate that the tools exhibit very low consistency, suggesting that they are complementary rather than conflicting. Moreover, the consistency almost universally decreases as versions are upgraded. Additionally, using the high-risk reentrancy vulnerability as a case study, we explore optimal tool combinations in terms of vulnerability coverage. Vandal performs the best tool because it almost covers all the vulnerable contracts reported by other tools.

Our study provides an in-depth discussion on the persistence of well-known vulnerabilities in newer versions. By analyzing both a representative vulnerability case and the semantic drift in contract code over time, we reveal that developers must remain cautious of seemingly secure code patterns. Moreover, directly reusing or applying security knowledge from earlier versions often requires substantial adaptation to be effective in newer contexts. In addition, the study offers practical recommendations to guide future research efforts based on the findings from the three RQs, including tool extension and improvement, rule refinement, and tool integration.

Contribution. In summary, the contributions of this paper are:

- We conducted a comprehensive empirical study using 22 static analysis tools on a representative sample dataset, and further evaluated 10 tools—those compatible with all versions—on the largest publicly available real-world source code dataset containing 251,340 unique contracts.
- We proposed three key characteristics—compatibility, adaptability, and consistency—to systematically evaluate the impact of Ethereum and Solidity evolution on static tools.
- We provide an in-depth discussion of a representative vulnerability case and the semantic drift observed in newer versions, along with practical recommendations to guide future research efforts.
- The complete dataset, experimental results, and manually validated labels are well-documented and publicly accessible [1], which facilitates future related research.

2 Background

In this section, we present the major upgrades to Ethereum and significant changes in the Solidity language. The smart contract source code written in Solidity is first compiled into bytecode, which consists of a series of opcodes, and then the bytecode is executed by the Ethereum Virtual Machine (EVM). Therefore, we primarily focus on the impact of Ethereum upgrades on low-level opcodes and the effect of Solidity changes on the high-level source code of contracts.

2.1 Ethereum Hard Forks

Ethereum has undergone 19 significant upgrades [21], commonly referred to as hard forks, as shown in the scatter plot in Figure 1. The first major version of Ethereum, “Frontier”, was released on 30 July 2015, in block number 1, and the

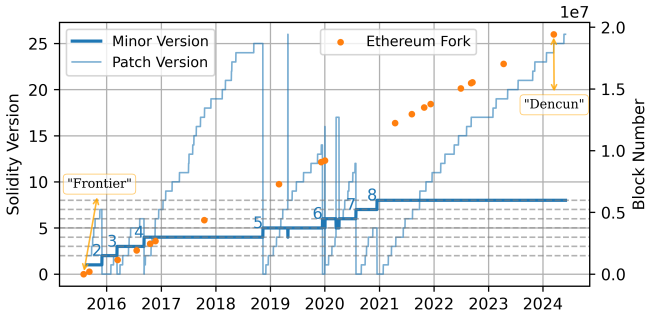

Fig. 1. Evolution of Ethereum (points, right vertical axis) and Solidity compiler versions (lines, left vertical axis).

Ethereum Fork	Date	EIP	New Opcode
Homestead	2016/03/14	EIP-7	DELEGATECALL
Byzantium	2017/10/16	EIP-140	REVERT
		EIP-211	RETURNDATASIZE
			RETURNDATACOPY
		EIP-214	STATICCALL
Petersburg	2019/02/28	EIP-145	SHL, SHR, SAR
		EIP-1014	CREATE2
		EIP-1052	EXTCODEHASH
Istanbul	2019/12/07	EIP-1344	CHAINID
London	2021/08/05	EIP-3198	BASEFEE
Shapella	2023/04/12	EIP-3855	PUSH0
Dencun	2024/03/13	EIP-1153	TLOAD, TSTORE
		EIP-5656	MCOPY
		EIP-7516	BLOBBASEFEE

Fig. 2. New opcodes.

Table 1. Comparison with related empirical evaluations of smart contract security tools.

Work	Venue	Dataset Format	Dataset Size	Dataset Version	Tools	Cross-Version Analysis
Durieux *et al.* [19]	ICSE'20	Source Code	48K	0.4.x–0.5.x	9	✗
Ghaleb *et al.* [24]	ISSTA'20	Source Code	50	0.5.x	6	✗
Ren *et al.* [46]	ISSTA'21	Source Code	46K	0.4.x–0.6.x	9	✗
Li *et al.* [36]	FSE'24	Source Code	9K	0.4.x–0.8.x	8	✗
Chaliasos *et al.* [12]	ICSE'24	DeFi Attacks	127	0.4.x–0.8.x	5	✗
Di Angelo *et al.* [17]	ESE'24	Bytecode	248K	0.4.x–0.8.x	12	✓
This work		Source Code	251K	0.4.x–0.8.x	22	✓

latest major version, "Dencun", was released on 13 March 2024, in block number 19,426,587. Each hard fork typically changes the rules of the Ethereum protocol based on Ethereum Improvement Proposals (EIPs) [20], which include core development, network specification, contract standard, and more. Among them, EIPs related to core development include additions and modifications of smart contract opcodes for the Ethereum Virtual Machine. For example, Figure 2 shows the new opcodes added during the Ethereum upgrades. These changes could lead to automatic static analysis tools that misinterpret or do not recognize the semantics of contract bytecode. Consequently, these tools might become unusable or report incorrect results.

2.2 Solidity Breaking Changes

Solidity versions follow Semantic Versioning [59], taking the form X.Y.Z, where X, Y, and Z respectively represent the major, minor, and patch versions [55]. The major version of Solidity remains at 0, while the minor version has changed to 0.8.x, with patch versions being released frequently, as shown in Figure 1. Since 0.4.x underwent updates to 0.5.x over an extended period spanning about two years, from 2016 to 2018, it remains a focal point for early smart contract security research [38] [57] [62]. Subsequently, within two years, Solidity swiftly updated from minor version 0.5.x to minor version 0.8.x. Each minor version update from 0.5.x to 0.8.x includes breaking changes, which potentially break backward compatibility with existing code bases.

2.3 Related Work

Our work is closely related to empirical studies that evaluate smart contract security analysis tools. As shown in Table 1, in the early years, empirical evaluations were limited to low-version datasets (no higher than 0.6.x). For example, Durieux *et al.* [19] evaluated the effectiveness of nine state-of-the-art analysis tools on 47,518 real-world contracts to recommend the optimal tool combination. Ghaleb *et al.* [24] proposed a bug injection approach to evaluate six existing

tools using generated buggy contracts. Ren *et al.* [46] evaluated the effectiveness of nine well-known testing tools on 45,622 real-world contracts to choose the best tool.

More recently, Li *et al.* [36] evaluated the detection capabilities of existing tools in the context of real-world high-profile attacks and surveyed practical requirements of practitioners. Chaliasos *et al.* [12] developed a comprehensive benchmark for evaluating static analysis tools. However, despite covering multiple Solidity versions, these studies did not specifically analyze tool performance across different contract versions.

In addition, several studies have begun exploring the impact of Ethereum and Solidity evolution on smart contract security tools. For example, Hwang *et al.* (2020) [31] focused on how smart contract vulnerabilities were affected by the release of version 0.5.x of the Solidity compiler. Then, Kado *et al.* (2023) [33] explored the impact of compiler updates on common vulnerabilities based on the analysis results of SmartCheck in different versions of the dataset. Similarly, Di Angelo *et al.* (2024) [17] analyzed the robustness of existing bytecode-based vulnerability detection tools as smart contracts evolve. However, these works [17, 31, 33] are restricted to evaluating a single version upgrade (0.5.x), focusing on a single tool (SmartCheck), or analyzing data at the bytecode level that constrained the exploration of underlying causes. None of them has provided a comprehensive evaluation and in-depth insights.

In contrast to the aforementioned work, we strive to systematically and comprehensively evaluate the performance of 22 available static analysis tools and investigate the underlying causes of their behavior across various contract versions. We focus on a source code dataset that facilitates a deeper understanding of contract semantics and enables an insightful analysis of the underlying causes. In addition, we propose an evaluation methodology based on three aspects, compatibility, adaptability, and consistency, to inspire future evaluations of similar tools.

3 Methodology

3.1 Research Questions

In this work, we aim to answer the following research questions:

RQ1. *[Compatibility] Do static tools execute successfully on every version of smart contracts?*
This question focuses on investigating whether static tools remain usable when applied to different versions of smart contracts. It explores the compatibility of these tools across diverse versions, considering factors such as changes in the Solidity language and updates to the Ethereum platform.

RQ2. *[Adaptability] How effective do different static tools perform across various versions?*
This question aims to evaluate the adaptability of static tools when applied to different versions. It evaluates the number (detection rate), accuracy (true positive rate) and effectiveness of vulnerabilities detected by different tools across various versions and uncovers potential causes.

RQ3. *[Consistency] How does the overlap of detection results between tools change across different versions? Which tool combinations yield the best effectiveness?*
This question is designed to investigate whether static tools yield consistent results and how these results vary with Solidity version upgrades. Specifically, it examines the overlap of vulnerable contracts detected by multiple tools and identifies the most effective combinations of tools for comprehensive coverage.

3.2 Tools Collection

To comprehensively cover existing static tools, we reviewed relevant studies that survey or evaluate security tools for smart contracts, gathering all the tools they summarized.

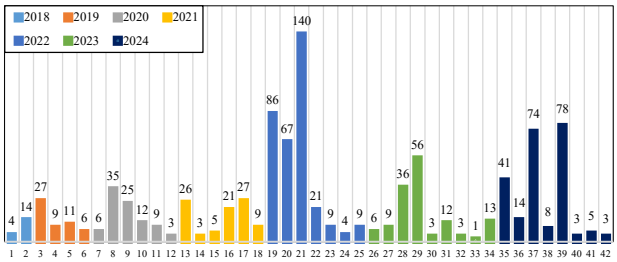

Fig. 3. The number of tools summarized in 41 papers.

Filter Criteria	Number of Excluded Tools	Number of Remaining Tools
C1	136	163
C2	83	80
C3	55	25
C4	5	20

Fig. 4. Tool Filtering.

Literature Review. We conducted a light systematic literature review following the guidelines reported by Stapić *et al.* [52]. We focus on the DBLP [47] literature database because it includes comprehensive research findings in the field of computer science [63]. We utilized a search query that targets titles containing "smart contract" along with any of the terms "survey", "review", "empirical", or "tool" for the years 2015 to 2024, resulting in 178 publications, among which 38 were non-peer-reviewed papers and were directly excluded. After meticulous manual screening, we identified 42 pertinent papers and gathered all the tools summarized in them as shown in Figure 3.

Tool Filtering. In total, we collected 299 unique security tools, which are a union of the tools studied in the 42 papers in Figure 3. We then filtered our target toolset according to the following four criteria. As shown in Figure 4, we ultimately filtered down to 20 static tools for evaluation.

- C1 The tool is designed to directly detect smart contract vulnerabilities on Ethereum. This excludes security-support tools such as decompilers, visualization tools, and style guiders. Vulnerability detection is the core security task and offers stronger comparability than these auxiliary tools.
- C2 The tool belongs to the category of static analysis tools. This study focuses exclusively on static analysis tools and therefore does not include dynamic and AI-based tools.
- C3 The tool is open-source. Open-source code provides the foundation for analyzing the underlying reasons behind tool failures or unexpected anomalies.
- C4 The tool provides a command-line interface, which facilitates the scalability of the analyzes.

In addition, we incorporate two extensions of teEther [35], namely eTainter [25] and AChecker [26], into our evaluation to enhance the completeness of the toolset. Consequently, the final evaluation encompasses 22 tools in total.

Evaluated Tools. We provide detailed information for each static tool evaluated in this work in Table 2. Most tools are derived from academic papers cited in the second column and open-sourced on GitHub. We show the year of academic publication and the version of the tool used in this work, which hints at the maintenance of the tool. The main approaches performed by the tools can be roughly divided into three categories: (1) Pattern Matching (PM), which includes regular expression matching, (2) Intermediate Representation Matching (IRM), and (3) Symbolic Analysis(SA), which includes symbolic execution and taint analysis.

3.3 Dataset Collection and Analysis

Dataset Collection. While several publicly available vulnerability datasets exist (e.g., CVE [16], SWC [49], $SB^{CURATED}$ [19], etc.), they are unsuitable for multi-version evaluation in this work due to four key limitations: (1) incomplete version coverage [19, 24, 46], (2) limited dataset size [12, 16, 49], (3) redundant experimental results [19], and (4) insufficient diversity within individual versions [36]. Therefore, similar to the work [17], we utilize a comprehensive real-world unlabeled dataset.

Table 2. Static tools evaluated in this work.

No.	Tool	URL	Year	Version	Approach[a]	Level[a]	Development Type
1.	Conkas	https://github.com/nveloso/conkas	2021-2021	#4e0f256	SA	BC	Original
2.	Ethainter [9]	https://doi.org/10.5281/zenodo.3760402	2020-2020	v0.1	SA	BC	Original
3.	eThor [48]	https://secpriv.wien/ethor/	2020-2023	v2023	SA	BC	Original
4.	HoneyBadger [61]	https://github.com/christoftorres/HoneyBadger	2019-2019	v0.0.1	SA	BC	Based on Oyente
5.	MadMax [28]	https://github.com/nevillegrech/MadMax	2018-2021	#6e9a6e9	IRM	BC	Based on Vandal
6.	Mythril	https://github.com/Consensys/mythril	2017-now	v0.23 ↑	SA	BC	Original
7.	Osiris [60]	https://github.com/christoftorres/Osiris	2018-2018	v0.0.1	SA	BC	Based on Oyente
8.	Oyente [38]	https://github.com/enzymefinance/oyente	2016-2017	v0.2.7	SA	BC	Original
9.	RA [14]	https://github.com/wanidon/RA	2020-2021	#69bf103	SA	BC	Original
10.	Securify [62]	https://github.com/eth-sri/securify	2018-2020	#51ba124	SA	BC	Original
11.	Securify2	https://github.com/eth-sri/securify2	2020-2021	#def1e30	PM	SC	Original
12.	Semgrep	https://github.com/Decurity/semgrep-smart-contracts	2023-now	v1.0.0 ↑	PM	SC	Original
13.	Slither [23]	https://github.com/crytic/slither	2019-now	v0.9.6 ↑	IRM	SC	Original
14.	SCA [31]	https://github.com/sjmini/icse2020-Solidity	2020-2020	#38576d6	PM	SC	Original
15.	SmartCheck [57]	https://github.com/smartdec/smartcheck	2018-2019	v2.0	IRM	SC	Original
16.	Solhint	https://github.com/protofire/solhint	2017-now	v3.3.8 ↑	PM	SC	Original
17.	SolidityCheck [64]	https://github.com/xf97/SolidityCheck	2019-2021	#a859f4b	PM	SC	Original
18.	teEther [35]	https://github.com/nescio007/teether	2018-2021	#04adf56	SA	BC	Original
19.	Vandal [10]	https://github.com/usyd-blockchain/vandal	2018-2020	#d2b0043	IRM	BC	Original
20.	VeriSmart [50]	https://github.com/kupl/VeriSmart-public	2020-2023	#36d191e	SA	SC	Original
21.	eTainter [25]	https://github.com/DependableSystemsLab/eTainter	2022-2023	#2735caf	SA	BC	Based on teEther
22.	AChecker [26]	https://github.com/DependableSystemsLab/AChecker	2022-2023	#e4e47ef	SA	BC	Based on teEther

[a] 'PM' for 'Pattern Matching' (Regular Expressions), 'IRM' for 'Intermediate Representation Matching', and 'SA' for 'Symbolic Analysis' (execution/taint). 'SC' for 'Source Code', and 'BC' for 'ByteCode'.

[b] ↑ denotes that the tool is continuously being updated.

Specifically, our dataset, Smart Contract Sanctuary - Ethereum (SCSE), originates from the GitHub open-source project Smart Contract Sanctuary [43] (1.5k Stars), which is maintained by the renowned blockchain company ConsenSys [15]. This repository continuously gathers all verified contract source codes for various blockchain networks using the contract Application Binary Interface (ABI) retrieved from blockchain browsers, such as Etherscan. We started by downloading all contracts from the Ethereum mainnet as of commit ID 575175c [44]. Next, we use the Grammars-v4 [34] to to flatten the source code of each contract address into a single file, facilitating efficient analysis. Then, we removed duplicate contract source code after eliminating spaces, tabs, and line breaks, following the approach of the work [46], to eliminate data redundancy. Note that, in this study, one contract is considered the complete contract source code associated with a unique address, rather than a `Contract` definition in Solidity. Finally, we obtained a dataset comprising a total of 251,340 contracts. Compared to existing publicly available datasets used in smart contract security research [19, 46, 65], which contain no more than 50,000 contracts, *this is the largest source code dataset to date.*

Furthermore, we constructed a sample dataset, SESC-Sample, to quantitatively assess the compatibility of tools across versions in § 4, without significant expenditure of execution time and computational resources. As demonstrated by the line graph in Figure 5, for each minor version (e.g., 0.4.x), we initially identified the patch version (e.g., 0.4.24) with the highest proportion of contracts, and subsequently randomly sampled contracts of this patch version, adhering to a 95% confidence level and a 5% confidence interval. Finally, we sampled 371, 355, 375, 361, and 381 contracts from versions 0.4.24, 0.5.12, 0.6.12, 0.7.6, and 0.8.7, respectively, totaling 1843 contracts, to form the dataset SESC-Sample.

Dataset Analysis. As shown in Figure 5, the stacked bar chart represents the number of contracts for each version in SCSE dataset. Among them, the number of contracts for version 0.8.x (177,537) is the highest, far exceeding the number for other versions, followed by version 0.4.x (31,972) and 0.6.x (20,827), and the number of contracts for version 0.5.x (10,944) and 0.7.x (10,060) is the lowest.

The differences in the number of contracts between versions are due to two main factors: (1) As shown in Figure 1, the time span for each version differs. Version 0.8.x has lasted 4 years, 0.4.x lasted 2 years, while 0.5.x, 0.6.x and 0.7.x

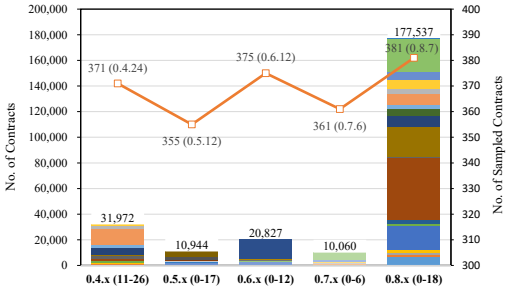

Fig. 5. The number of contracts of different versions in SCSE (bars, left vertical axis) and SCSE-Sample (line, right vertical axis) datasets.

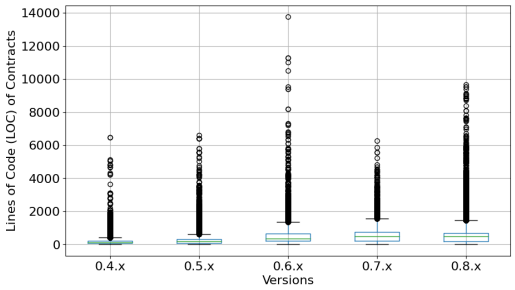

Fig. 6. Boxplot of LOC grouped by versions.

spanned only around one year. (2) As reported by Etherscan [22], the number of verified contracts has surged rapidly in recent years. Moreover, the large number of version 0.8.x contracts indicates that developers are more widely adopting the latest compiler versions, which contradicts previous research claims [56] that the majority of contracts are developed using lower versions of the compilers as they are more familiar to developers. This difference can also be observed in a recent large-scale study [13], which collected 6,454 real-world audit reports and revealed a similar contract version distribution: v0.4+ (4%), v0.5+ (7%), v0.6+ (24%), v0.7+ (6%), and v0.8+ (59%). These observations draw our attention to the inherently imbalanced distribution of contract versions in the real world, emphasizing the importance of constructing evaluation datasets that reflect realistic usage rather than artificially balanced benchmarks.

To understand changes in contract complexity across different versions, we used the open-source tool CLOC [5] to count the lines of code (LOC) of contracts in the SCSE dataset. Figure 6 illustrates the LOC distribution for different versions of the contracts. The average LOCs for different versions are as follows: 190 for 0.4.x, 340 for 0.5.x, 490 for 0.6.x, 607 for 0.7.x, and 522 for 0.8.x, indicating that the contracts become more complicated with each version update. Although the vast majority of contracts have fewer than 1,000 lines of code, there is still a considerable number (8%) of contracts that far exceed 1,000 lines.

4 RQ1: Compatibility of Static Tools

In this section, the compatibility of static tools is evaluated on five minor `Solidity` versions from 0.4.x to 0.8.x. The compatibility evaluation aims to capture the *dynamic usability* of tools across Solidity versions, encompassing both functionality and scalability aspects, including failures such as out-of-time and out-of-memory errors.

4.1 Experimental Setup.

In the experiment, the evaluated tools include the previously mentioned 22 static tools mentioned above as shown in Table 2, and the data set utilized is the SESC-Sample as shown in § 3.3 to efficiently evaluate the tools. We use SmartBugs2 [18] (v2.0.8), a state-of-the-art extensible platform with a uniform interface to contract analysis tools, to aid in the rapid deployment of environments and obtain results in a unified format to streamline the analysis process.

The execution time for analyzing a contract is limited to 2 hours for each tool. However, since more than half of the contracts analyzed by RA (Re-entrancy Analyzer) [14] exceed the 2-hour timeout, we analyze the performance metric in its paper [14] (0.2 seconds per byte of code length). Based on this, we estimate a maximum runtime of 18 hours for the largest contracts in our dataset. To accommodate this, we extend the timeout from 2 hours to 20 hours, providing a

No. Tool	0.4.x	0.5.x	0.6.x	0.7.x	0.8.x	Compatibility
1. Conkas	89%	75%	0%	0%	0%	✗ 2 / 5
2. Ethainter	99%	99%	99%	98%	100%	✓ 5 / 5
3. eThor	69%	3%	0%	0%	0%	✗ 1 / 5
4. HoneyBadger	95%	0%	0%	0%	0%	✗ 1 / 5
5. MadMax	99%	99%	99%	98%	100%	✓ 5 / 5
6. Mythril	100%	98%	94%	76%	100%	✓ 5 / 5
7. Osiris	89%	0%	0%	0%	0%	✗ 1 / 5
8. Oyente	98%	0%	0%	0%	0%	✗ 1 / 5
9. RA	49%	49%	49%	43%	46%	✗ 0 / 5
10. Securify	92%	81%	46%	39%	39%	✗ 2 / 5
11. Securify2	0%	75%	10%	0%	2%	✗ 1 / 5
12. Semgrep	98%	100%	99%	100%	100%	✓ 5 / 5
13. Slither	100%	99%	100%	100%	99%	✓ 5 / 5
14. SCA	100%	80%	8%	12%	2%	✗ 2 / 5
15. SmartCheck	99%	100%	99%	100%	100%	✓ 5 / 5
16. Solhint	100%	100%	100%	100%	100%	✓ 5 / 5
17. SolidityCheck	100%	100%	100%	100%	100%	✓ 5 / 5
18. teEther	80%	76%	32%	38%	40%	✗ 2 / 5
19. Vandal	100%	100%	100%	100%	99%	✓ 5 / 5
20. VeriSmart	92%	97%	96%	96%	98%	✓ 5 / 5
21. eTainter	88%	77%	33%	35%	40%	✗ 2 / 5
22. AChecker	24%	26%	9%	17%	39%	✗ 0 / 5
Total (≥50%)	19/22	16/22	10/22	10/22	10/22	10/22 (45%)

Fig. 7. The execution success rate of static tools on SESC-Sample dataset.

No. Tool	Execution Exceptions	0.4.x	0.5.x	0.6.x	0.7.x	0.8.x
1. Conkas	C1: Parser (solidity_parser)	0%	1%	86%	76%	96%
	C2: Compilation (solcx)	2%	17%	3%	6%	4%
	C3: Source mappings	0%	0%	11%	19%	0%
	C7: Tool-specific infrequent bugs	9%	6%	0%	0%	0%
3. eThor	C5: Unsupported DELEGATECALL	1%	18%	1%	5%	37%
	C5: Encountered an unknown bytecode	1%	77%	95%	90%	56%
	C6: Out of time	16%	0%	1%	1%	0%
	C6: Out of memory	12%	1%	1%	0%	0%
	C7: Tool-specific infrequent bugs	1%	1%	2%	4%	7%
4. HoneyBadger	C1: Unsupported version	0%	100%	100%	94%	100%
	C6: Out of time	5%	0%	0%	0%	0%
	C7: Tool-specific infrequent bugs	0%	0%	0%	6%	0%
7. Osiris	C1: Unsupported version	0%	97%	96%	100%	100%
	C6: Out of time	10%	0%	0%	0%	0%
	C7: Tool-specific infrequent bugs	1%	4%	3%	0%	0%
8. Oyente	C3: Source mappings	0%	0%	0%	0%	99%
	C5: POP implementation	0%	25%	43%	40%	0%
	C5: Unknown SHR and INVALID	0%	74%	56%	60%	0%
	C7: Tool-specific infrequent bugs	2%	1%	1%	1%	1%
9. RA	C5: Unknown CODEDATACOPY	2%	1%	3%	6%	39%
	C5: Unknown STATICCALL	0%	8%	15%	17%	1%
	C5: Opcode implementation	17%	35%	18%	24%	8%
	C6: Out of time	29%	4%	5%	5%	5%
	C7: Tool-specific infrequent bugs	3%	2%	9%	5%	1%
10. Securify	C1: Decompilation	2%	1%	3%	4%	31%
	C4: Dataflow analysis	6%	14%	48%	50%	28%
	C6: Out of time	0%	5%	2%	7%	2%
11. Securify2	C2: Abstract syntax tree matching	94%	14%	87%	100%	98%
	C4: Control flow graph compilation	6%	0%	2%	0%	0%
	C7: Tool-specific infrequent bugs	0%	12%	1%	0%	0%
14. SCA	C2: Abstract syntax tree matching	0%	17%	0%	0%	0%
	C2: Parser	0%	2%	92%	88%	98%
18. teEther	C6: Out of time	6%	2%	2%	3%	7%
	C6: Out of memory	7%	20%	61%	51%	41%
	C7: Tool-specific infrequent bugs	7%	2%	6%	8%	12%
21. eTainter	C6: Out of time	4%	1%	3%	3%	0%
	C6: Out of memory	6%	21%	60%	54%	50%
	C7: Tool-specific infrequent bugs	2%	2%	4%	7%	10%
22. AChecker	C6: Out of time	68%	52%	23%	22%	1%
	C6: Out of memory	7%	21%	63%	55%	51%
	C7: Tool-specific infrequent bugs	2%	1%	5%	7%	9%

Fig. 8. The execution exceptions thrown by tools that lack compatibility.

margin of safety for rare edge cases. The memory usage for the execution of the tools is not limited, and the tools are executed with their default parameters.

All experiments are conducted on four machines: one running Ubuntu 18.04 LTS with 48 Intel(R) Xeon(R) Gold 6226 CPUs @ 2.70 GHz and 187 GB of memory, and three running Ubuntu 20.04 LTS, equipped with 96, 64, and 72 CPUs and 376, 376, and 251 GB of memory, respectively.

4.2 Experimental Results.

Figure 7 presents the execution success rate of static tools on the SESC-Sample dataset. For a contract, if the tool is able to output its detection results, the execution is considered successful. The execution success rate of a tool is calculated as the number of successfully executed contracts divided by the total number of contracts for a specific version. If the execution success rate of a tool in a certain version exceeds 50% , the tool is considered *compatible* with that version.

As indicated in the last column of Figure 7, all tools, except for RA and AChecker, are compatible with at least one version. The execution success rate of RA is approximately 50% across all versions, suggesting that its compatibility is not impacted by version updates but rather by its own poor reliability and frequent bugs. AChecker exhibits low execution success rates across versions with substantial variation, indicating limited compatibility to language changes.

Besides, the higher the version, the fewer tools are compatible. For example, only Securify2 completely lacks support for version 0.4.x (0%), as it is an improvement over Securify, designed primarily to detect contracts in version 0.5.x and beyond. Moreover, Figure 7 shows that if a tool is compatible with 0.6.x, it is also compatible with 0.7.x and 0.8.x, which implies that poorly compatible tools are stuck on 0.4.x or 0.5.x and advanced tools will always follow the latest version. In total, 19, 16, 10, 10, and 10 out of the 22 tools support Solidity versions 0.4.x through 0.8.x, respectively. Together, only 45% (10/22) of the static tools are compatible with all versions.

Finding 1: Merely 45% (10 out of 22) of static analysis tools are fully compatible with all Solidity versions ranging from 0.4.x to 0.8.x.

4.3 Reason Analysis.

As shown in Table 2, four tools (Mythril, Slither, Semgrep, and Solhint) are continuously maintained and developed by commercial security companies or dedicated teams. It is not surprising that they are compatible with all versions. Interestingly, the remaining six compatible tools, although developed by academic teams and no longer actively maintained, still demonstrate good compatibility. For example, SmartCheck leverages ANTLR4 [6] to parse source code, which provides error recovery capabilities that allow it to tolerate partially mismatched syntax structures. This makes the tool resilient to certain syntactic deviations in newer Solidity versions. Notably, as shown in Table 2, HoneyBadger and Osiris are extensions of Oyente, MadMax is developed based on Vandal, while eTainter and AChecker are developed based on teEther. These tools exhibit similar compatibility to their respective base tools, suggesting that they may inherit the same underlying limitations.

In addition, as shown in Figure 8, we analyze the execution exceptions thrown by 12 static tools that lack compatibility, and provide the proportion of each exception in different versions. Following the workflow design and implementation of each static tool, the execution exceptions affecting tool compatibility fall into the following seven categories (C1-C7), with the first six being related to version updates, while the last one is not.

- C1: (De)compiler execution (HoneyBadger, Osiris, and Securify). HoneyBadger and Osiris are improvements based on Oyente, but they have the compiler (solc) version restrictions, explicitly stating that they do not support contracts of version 0.5.x and above (>90%). For version 0.8.x, 31% of the exceptions in Securify are caused by decompilation errors.
- C2: Source code parser or abstract syntax tree matching (Conkas, Securify2, and SCA). The main reason Conkas is incompatible with 0.6.x to 0.8.x is due to the parser error (>70%). This incompatibility arises from its reliance on outdated Python packages: `solidity-parser` [58] and `solcx` [7]. These packages are used to identify contract versions and compile contracts to obtain runtime bytecode, respectively, but are not compatible with parsing versions 0.6.x and above. We find that `solcx` continues to be maintained, whereas `solidity-parser` stopped updates in November 2021. Securify2 only implements matching for the abstract syntax tree structure of version 0.5.x. SCA implements lexical analysis based on the Scala programming language, which is not adapted to analyze new literals of Solidity in 0.6.x (92%) and above.
- C3: Source mappings (Conkas, Oyente). Conkas leverages the source mappings in Solidity to obtain the runtime bytecode of each contract for subsequent symbolic analysis. However, it encounters execution failures during the source code to bytecode mapping process in 0.6.x (11%) and 0.7.x (19%). This issue primarily arises because, starting from version 0.6.x, the source mapping structure in Solidity introduces a new field "`m`", which denotes

the "modifier depth"[51]. However, the analysis of source mappings implemented by Conkas is not compatible with this change. For version 0.8.x, 99% of the exceptions in Oyente are caused by the same issue.

C4: Data flow and Control flow graph analysis (Securify, Securify2). Securify analyzes contract bytecode, but its data flow analysis rules have not been adapted to handle the new bytecode introduced in 0.6.x (48%) and above. Securify2 encounters 6% control flow graph compilation errors in 0.4.x.

C5: Bytecode or opcode implementation (eThor, Oyente, RA). The main reason eThor, Oyente and RA are incompatible with higher versions ($\geq$ 0.5.0) is their failure to implement new opcodes or update certain opcode implementations, which hinders their ability to perform bytecode-level symbolic analysis.

C6: Performance in terms of time or memory (teEther, eTainter, and AChecker). The compatibility of these tools is mainly limited by the memory performance in 0.6.x ($\geq$60%) and higher.

C7: Tool-specific infrequent bugs, such as list index out-of-bounds errors, invalid computational operations within the Z3 solver, unknown errors, and so on.

In conclusion, the impact of each exception category on tool compatibility varies in severity. We further quantified the number of tools for which each exception category accounts for at least 50% of the failed executions. Among them, (de)compiler execution exceptions (C1) and bytecode/opcode implementation exceptions (C5) have the strongest influence, acting as primary factors for three tools each—Conkas, HoneyBadger, and Osiris for C1; eThor, Oyente, and RA for C5. Performance issues (C6) are the main limiting factor for another three tools: teEther, eTainter, and AChecker. Source code parser and abstract syntax tree matching exceptions (C2) are also significant, critically affecting Securify2 and SCA. In contrast, exceptions related to source mappings (C3) and dataflow/control-flow graph analysis (C4) primarily impact only one tool each—Oyente for C3 and Securify for C4. Tool-specific infrequent bugs (C7), however, have a negligible impact on compatibility.

Finding 2: The primary cause of tool incompatibility can be attributed to changes in language specifications, compiler output, and the increasing complexity of contracts.

5 RQ2: Adaptability of Static Tools

In this section, we evaluate the adaptability of static tools on different versions of smart contracts. Adaptability is intended to capture the *dynamic effectiveness* of a tool throughout the evolution of smart contracts. To ensure reliable full-version and large-scale analysis, we focus on the 10 tools that are compatible with all versions, as identified in RQ1.

5.1 Experimental Setup.

According to RQ1 in § 4, ten static analysis tools are evaluated as compatible with all versions. To perform a comprehensive evaluation of all versions, we conduct experiments focusing on these ten tools using the large-scale SESC dataset shown in Figure 5. The evaluation on the large-scale dataset, which reflects the real-world distribution of smart contract versions, provides valuable insights into the practical effectiveness of tools. To mitigate potential bias caused by the data imbalance across versions, all analyses are conducted per version and use percentage-based metrics (e.g., detection rates) rather than absolute counts. The execution time, memory, and parameter settings of each tool align with those of RQ1 as shown in § 4.1, and all experiments are carried out on four high-performance machines introduced in RQ1.

Vulnerability Selection. Our selection criteria for vulnerabilities consider two aspects. First, the severity of the vulnerability, such as the extent of asset losses it has caused in real-world incidents. Second, the support for detection by most tools, which not only indicates the prevalence of the vulnerability but also facilitates comparative analysis

Table 3. High-risk vulnerabilities and static detection tools.

DASP/SWC	Vulnerability	Ethainter	MadMax	Mythril	Semgrep	Slither	SmartCheck	Solhint	SolidityCheck	Vandal	VeriSmart
Top 1 / SWC-107	Reentrancy (REN)			✓	✓	✓		✓		✓	
Top 2 / SWC-106	Unprotected SELFDESTRUCT Instruction (USI)	✓		✓	✓	✓				✓	✓
Top 2 / SWC-115	Authorization through tx.origin (TXO)			✓		✓	✓	✓	✓	✓	
Top 2 / SWC-105	Unprotected Ether Withdrawal (UEW)	✓		✓	✓	✓				✓	✓
Top 3 / SWC-101	Integer Overflow and Underflow (IOU)		✓	✓	✓		✓				✓

[a] ✓ denotes that the tool supports the detection of a specific vulnerability.
[b] ✓ indicates that there is an anomaly in the tool's detection rate trend according to Table 4.

among different tools. Specifically, we first analyze the types of vulnerabilities supported by each tool and find that the vulnerabilities detected by at least half of the tools correspond precisely to the top three high-risk categories in the the Decentralized Application Security Project (DASP) [40] ranking. As shown in Table 3, this includes five specific vulnerability types defined by Smart Contract Weakness Classification (SWC) [49]. This finding suggests that these high-risk vulnerabilities are the primary focus of current tools and represent the key vulnerabilities in our evaluation.

5.2 Detection Results

To answer RQ2, we analyze trends in the detection rate of tools in different versions to evaluate their adaptability. *We assume that if the detection rate of a tool exhibits a trend opposite to that of other tools in different versions, it suggests that the tool is not properly adapting to changes in vulnerabilities introduced by update versions.*

Detection Rate. For any version V belonging to {0.4.x, 0.5.x, 0.6.x, 0.7.x, 0.8.x}, and any tool T belonging to {Ethainter, MadMax, Mythril, Semgrep, Slither, SmartCheck, Solhint, SolidityCheck, Vandal, VeriSmart }, the detection rate of tool T on version V is calculated using Equation 1.

$$Detection\ Rate_{T,V} = \frac{The\ number\ of\ vulnerable\ contracts\ reported\ by\ tool\ T\ on\ version\ V}{The\ total\ number\ of\ contracts\ on\ version\ V} \quad (1)$$

Detection Results. Table 4 shows the detection rate of ten static tools that are compatible with all five versions, for each vulnerability and version. The first column lists the type of vulnerability and the second column provides the tool name. The subsequent five columns display the detection rate of the tools. Each cell represents the detection rate of a specific tool in a particular version. The last row of each vulnerability represents the average detection rate of the tools. In each cell, the symbols ▲, ▼, and ■ denote an increase, decrease, or no change in the detection rate relative to the previous version. Symbols highlighted in red indicate anomalous behavior. The last column shows the overall detection rate of the tools.

Detection Rate Trend. On average, as versions are upgraded, the detection rates for REN and TXO show a gradual increase, while the rate for IOU decreases. USI exhibits a significantly higher rate in 0.4.x and 0.7.x, whereas UEW shows a markedly lower rate in 0.5.x. To explore the underlying causes of these trends, we analyze the ratio of vulnerability-related static code features across different versions, as shown in Table 5. The detection rate trends for REN, UEW, USI, and TXO are consistent with the distribution trends of the associated static code features. This helps justify the changes in detection rate trends. In particular, while the code features related to IOU are broadly distributed, the decline in the IOU detection rate is understandable due to factors such as the widespread use of SafeMath library [42] and the default overflow checks of arithmetic operations [54] implemented by the Solidity compiler starting from 0.8.x.

Table 4. The detection rate of ten static tools that are compatible with all five versions.

Vulnerability	Tool	0.4.x	0.5.x	0.6.x	0.7.x	0.8.x	Total
REN	Mythril	■ 7.94%	▼ 1.17%	▲ 2.07%	▼ 2.06%	▲ 5.82%	5.42%
	Semgrep	■ 0.10%	▲ 4.87%	▼ 3.61%	▲ 13.14%	▲ 17.19%	13.19%
	Slither	■ 41.48%	▼ 28.26%	▲ 59.80%	▲ 66.23%	▲ 88.61%	76.70%
	Solhint	■ 14.89%	▼ 1.78%	▲ 3.48%	▲ 4.28%	▼ 2.07%	3.89%
	Vandal	■ 62.31%	▼ 34.94%	▲ 52.84%	▲ 67.70%	▼ 67.49%	64.21%
	Average	■ 25.34%	▼ 14.21%	▲ 24.36%	▲ 30.68%	▲ 36.23%	32.68%
USI	Ethainter	■ 0.40%	▼ 0.22%	▼ 0.10%	▲ 0.87%	▼ 0.14%	0.20%
	Mythril	■ 0.37%	▼ 0.05%	▼ 0.03%	▲ 0.19%	▼ 0.08%	0.12%
	Semgrep	■ 3.1e-5	▼ 0.00%	■ 0.00%	■ 0.00%	▲ 1.7e-5	1.6e-05
	Slither	■ 0.29%	▼ 0.15%	▼ 0.04%	▲ 0.25%	▼ 0.04%	0.09%
	Vandal	■ 0.46%	▼ 0.06%	▲ 0.19%	▲ 0.43%	▼ 0.16%	0.21%
	VeriSmart	■ 1.36%	▼ 0.29%	▼ 0.14%	▲ 0.94%	▼ 0.04%	0.26%
	Average	■ 0.48%	▼ 0.13%	▼ 0.08%	▲ 0.45%	▼ 0.08%	0.15%
TXO	Mythril	■ 1.18%	▼ 0.21%	▲ 0.80%	▼ 0.48%	▲ 2.08%	1.71%
	Slither	■ 0.15%	▼ 0.11%	▲ 1.81%	▼ 0.85%	▲ 5.31%	3.95%
	SmartCheck	■ 0.75%	▲ 1.25%	▲ 1.66%	▲ 3.21%	▼ 2.33%	2.06%
	Solhint	■ 1.53%	▲ 2.30%	▲ 4.84%	▼ 4.37%	▲ 10.46%	8.26%
	SolidityCheck	■ 1.53%	▼ 1.43%	▲ 4.09%	▼ 1.89%	▲ 9.88%	7.65%
	Vandal	■ 1.49%	▼ 1.17%	▲ 2.48%	▼ 2.10%	▲ 5.74%	4.58%
	Average	■ 1.10%	▼ 1.08%	▲ 2.61%	▼ 2.15%	▲ 5.97%	4.70%
UEW	Ethainter	■ 0.61%	▼ 0.33%	▲ 0.70%	▲ 1.35%	▼ 0.37%	0.47%
	Mythril	■ 1.53%	▼ 0.57%	▼ 0.23%	▲ 0.57%	▼ 0.22%	0.41%
	Semgrep	■ 0.00%	▲ 0.19%	▼ 0.17%	▼ 0.00%	▲ 2.8e-5	0.02%
	Slither	■ 3.91%	▼ 3.36%	▲ 10.85%	▼ 9.44%	▲ 12.70%	10.89%
	Vandal	■ 4.04%	▼ 0.76%	▼ 0.55%	▲ 0.71%	▼ 0.25%	0.80%
	VeriSmart	■ 14.33%	▼ 3.79%	▲ 17.36%	▼ 17.12%	▼ 10.86%	11.78%
	Average	■ 4.07%	▼ 1.50%	▲ 4.98%	▼ 4.86%	▼ 4.07%	4.06%
IOU	MadMax	■ 1.15%	▼ 0.20%	▲ 0.56%	▲ 0.93%	▼ 0.46%	0.56%
	Mythril	■ 18.93%	▼ 3.09%	▲ 8.93%	▼ 5.72%	▼ 0.00%	3.51%
	Semgrep	■ 0.04%	▲ 0.09%	▲ 0.39%	▼ 0.31%	▼ 0.13%	0.14%
	SmartCheck	■ 0.81%	▼ 0.00%	■ 0.00%	■ 0.00%	■ 0.00%	0.10%
	VeriSmart	■ 77.41%	▼ 71.38%	▼ 61.25%	▼ 35.89%	▼ 3.24%	21.76%
	Average	■ 19.67%	▼ 14.95%	▼ 14.22%	▼ 8.57%	▼ 0.76%	5.21%

[a] ■ denotes that the detection rate remains unchanged compared to the previous version. ▲ denotes an increase in the detection rate relative to the previous version. ▼ denotes an decrease in the detection rate relative to the previous version. Symbols highlighted in red indicate anomalous behavior.

Table 5. Distribution statistics of code features in dataset.

Code Feature	Related Vulnerability	0.4.x	0.5.x	0.6.x	0.7.x	0.8.x	Total
CALL	REN, UEW	■ 75.99%	▼ 41.44%	▲ 57.15%	▲ 72.50%	▼ 71.67%	▼ 69.74%
SELFDESTRUCT	USI	■ 9.27%	▼ 5.65%	▼ 5.41%	▲ 7.41%	▼ 5.17%	▲ 5.82%
tx.origin	TXO	■ 1.87%	▲ 2.43%	▲ 5.03%	▼ 4.70%	▲ 10.85%	▼ 8.61%
ADD,SUB, MUL, DIV	IOU	■ 99.05%	▼ 97.38%	▲ 98.09%	▼ 97.87%	▲ 98.97%	▼ 98.79%

Finding 3: The detection rates consistently decrease in version 0.5.x, but begin to fluctuate in subsequent versions, indicating that the detection capability of the tools is not stable across different Solidity versions.

Adaptability of Tools according to Detection Rate. We evaluate the adaptability of each tool based on the relative trend of its detection rate across versions. A tool is considered to exhibit *anomalous behavior* if its trend deviates from that of the majority. In cases where tools are evenly divided, we refer to the average trend to guide our judgment. Given the absence of ground truth, this analysis offers a complementary perspective rather than a definitive conclusion. As shown in Table 4, compared to the average trend, at least two tools exhibit anomalous trends in certain versions for each type of vulnerability. In the following, we provide a detailed analysis of the tool's adaptability for each type of vulnerability.

REN The average detection rate significantly decreases in 0.5.x (14.21%) and then increases consistently. However, Semgrep exhibits anomalous trends in 0.5.x and 0.6.x. We speculate that this could be attributed to its release in 2022, which led to a focus on higher versions, resulting in insufficient detection capabilities for lower versions, such as 0.4.x (0.10%). Solhint exhibits an anomalous decrease in 0.8.x because it is restricted to detecting reentrancy risks associated with the built-in `transfer/send` functions, which are infrequently used in 0.8.x. Moreover, Mythril (0.01%) and Vandal (0.21%) show slight anomalous decreases in 0.7.x and 0.8.x, respectively, without significant underlying causes.

USI The average detection rate decreases from 0.5.x (0.13%), increases in 0.7.x (0.45%), and then decreases again in 0.8.x (0.08%). However, Semgrep detects very few vulnerabilities in 0.4.x and 0.8.x, and none in other versions, indicating a poor detection ability for USI. Moreover, Vandal exhibits an anomalous increase in 0.6.x, which may be due to a decline in accuracy and a corresponding increase in false positives.

TXO The trend in the average detection rate shows that 0.4.x and 0.5.x (~1%) are similar and lower than 0.6.x and 0.7.x (~2%/3%), while 0.8.x (~6%) exhibits a significant increase. However, SmartCheck shows anomalous increases in 0.5.x and 0.7.x, and an anomalous decrease in 0.8.x, indicating that its rules are poorly adapted to versions beyond 0.4.x. Moreover, Solhint exhibits an anomalous increase in 0.5.x, likely due to reduced accuracy.

UEW The average detection rate significantly decreases in 0.5.x (1.5%) and remains similar across other versions (~4%/5%). However, with the exception of VeriSmart, all other tools exhibit anomalous trends in at least one version, suggesting that version upgrades cause considerable fluctuations in UEW detection. The rationale might be that the UEW patterns are more diverse compared to other vulnerabilities (e.g., TXO), resulting in significant instability in the effectiveness of the tool across different versions.

IOU The average detection rate gradually decreases with each version update. However, MadMax exhibits an anomalous increase in version 0.7.x. Similar to its performance on the REN vulnerability, Semgrep shows poor detection capability in versions 0.4.x and 0.5.x. Since version 0.5.x, SmartCheck has failed to detect any vulnerable contracts, indicating a complete loss of detection capability. Moreover, while most tools demonstrate an increasing trend in version 0.6.x, VeriSmart shows a contrasting decrease.

Summary. As shown in Table 3, except for SolidityCheck and VeriSmart, all other tools (8/10) demonstrate a lack of adaptability to version updates when detecting a particular vulnerability. Despite this, VeriSmart (USI, UEW, IOU) and Slither (REN, USI, TXO) perform the best, showing stable adaptability across three vulnerabilities, while Mythril (USI, TXO) demonstrates stable adaptability for two vulnerabilities, and Ethainter (USI), SolidityCheck (TXO) and Vandal (TXO) show stable adaptability for one vulnerability.

> ***Finding 4:*** In terms of detection rate trend consistency, 9 out of 10 static analysis tools demonstrate unstable behavior, indicating limited adaptability to certain vulnerabilities in particular contract versions.

5.3 Manual Confirmation of True Positive Results

To further answer RQ2 based on the actual accuracy of the tools, we manually confirm the true positive (TP) rate in the detection results of § 5.2. In general, if a tool exhibits significant variations in TP rates across different versions, it suggests that the tool is influenced by version upgrades and has poor adaptability. In our evaluation, the differences in TP rates between different versions are represented by the standard deviation (σ).

Sampling Process. Given that the total number of vulnerable contracts detected by all tools combined is 610,173 across five types of vulnerabilities, a full manual audit is impractical. Thus, we adopt a random sampling approach for

Table 6. Manual audit rules and true positive confirmation.

Vulnerability	Audit Rules
REN	R_1: Contains executable external calls that can be invoked concurrently by any caller. R_2: The address for the external call can be any contract or account. R_3: The external call is followed by statements that modify non-local variables or make additional external calls.
USI	R_1: Contains executable built-in `suicide/selfdestruct` functions. R_2: `suicide/selfdestruct` functions can be executed successfully by any caller. R_3: The recipient address of `suicide/selfdestruct` is the caller (`msg.sender`/`tx.origin`) or can be modified arbitrarily by the caller.
TXO	R_1: The global variable `tx.origin` is used in executable conditional statements (e.g., `require` and `if`) for authentication. R_2: The current caller (`msg.sender`) is not an account address but can be any contract address.
UEW	R_1: Contains Ether withdrawal operations (e.g., `call/send/transfer` function calls), which can be successfully executed by any caller in the current contract. R_2: The recipient address of the withdrawal operation is the caller (`msg.sender`/`tx.origin`) or can be modified arbitrarily by the caller. R_3: The amount of the withdrawal operation can be changed to any value greater than the call payment (`msg.value`) or equal to the balance of the contract.
IOU	R_1: Contains executable integer arithmetic operations (e.g., `+, -,* ,/ ,** ,++ ,-, +=, -=, *=, /=`). R_2: The operands can be modified by the caller to any value. R_3: The results of arithmetic operations are not checked for integer overflow and underflow.

manual validation. To ensure statistical representativeness and robustness within each version, we perform independent random sampling for contracts in each version. Specifically, for the vulnerable contracts detected by each tool for each version, if the number of vulnerable contracts is less than or equal to 100, we manually confirm all results. Otherwise, we sample the vulnerable contracts, using a 90% confidence level and a 10% confidence interval, as done in [37].

Furthermore, when calculating the overall TP rate, we account for the varying contract volumes across versions as shown in Figure 5. To prevent distribution imbalance caused by independent sampling, we employ a weighted average approach to balance the overall TP rate. Specifically, the weight of each version is determined by its proportion of the total contract count, and the overall TP rate is computed as the weighted sum of the TP rates of each version.

Manual Audit Process. For each type of vulnerability, two authors with at least three years of experience in smart contract research verify each sampled vulnerable contract. To ensure reproducibility, manual audit rules are well-defined based on the description of vulnerabilities in SWC, as shown in Table 6. In cases of disagreement, a third author with five years of experience in the field is consulted to resolve the conflict. In general, manual confirmation of the total of 6,953 sampled contracts took approximately three months, and the resulting labels are publicly available in the GitHub repository [1] and subject to peer validation.

True Positive Results. Table 7 shows the manually confirmed TP rate of the tools. The first column lists the type of vulnerabilities, and the second column provides the tool name. The subsequent five columns display the TP rate of tools across different versions. Each cell represents the number of contracts manually confirmed, the count of true positives within them, and the corresponding TP rate. The last two columns represent the weighted total TP rate for a specific tool and the standard deviation of TP rates across different versions, respectively.

In total, the TP rate for most tools remains below 30%. Although this strongly highlights the poor accuracy of current static analysis tools, it aligns with existing research findings. For example, empirical study [19] found that the precision of each evaluated tool in the vulnerability dataset of ground truth does not exceed 30%. Furthermore, studies [45] and

Table 7. The manually confirmed TP rate of the tools, along with the standard deviation (σ) of TP rates across different versions.

	Tool	0.4.x[a]	0.5.x[a]	0.6.x[a]	0.7.x[a]	0.8.x[a]	Weighted Total[b]	σ[c]
REN	Mythril	3/66 (4.55%)	3/45 (6.67%)	5/59 (8.47%)	6/52 (11.54%)	3/68 (4.41%)	5.15%	2.98%
	Semgrep	0/31 (0.00%)	0/61 (0.00%)	9/63 (14.29%)	0/65 (0.00%)	**36/68 (52.94%)**	**38.58%**	**22.93%**
	Slither	**5/68 (7.35%)**	**6/67 (8.96%)**	3/68 (4.41%)	4/67 (5.97%)	2/68 (2.94%)	4.01%	2.37%
	Solhint	0/67 (0.00%)	0/51 (0.00%)	1/62 (1.61%)	0/59 (0.00%)	2/67 (2.99%)	2.24%	1.35%
	Vandal	4/68 (5.88%)	5/67 (7.46%)	**11/68 (16.18%)**	**8/67 (11.94%)**	14/68 (20.59%)	17.43%	6.09%
USI	Ethainter	4/45 (8.89%)	0/24 (0.00%)	0/21 (0.00%)	1/88 (1.14%)	2/53 (3.77%)	3.84%	3.76%
	Mythril	4/44 (9.09%)	0/5 (0.00%)	0/7 (0.00%)	0/19 (0.00%)	3/47 (6.38%)	5.67%	4.34%
	Semgrep	**1/1 (100.00%)**	0/0 (0.00%)	0/0 (0.00%)	0/0 (0.00%)	**3/3 (100.00%)**	**83.36%**	**54.77%**
	Slither	8/94 (8.51%)	0/16 (0.00%)	0/8 (0.00%)	**3/25 (12.00%)**	23/78 (29.49%)	22.39%	12.10%
	Vandal	0/47 (0.00%)	0/7 (0.00%)	0/40 (0.00%)	0/43 (0.00%)	2/56 (3.57%)	2.52%	1.60%
	VeriSmart	2/59 (3.39%)	0/32 (0.00%)	**1/30 (3.33%)**	3/95 (3.16%)	4/63 (6.35%)	5.32%	2.25%
TXO	Mythril	2/58 (3.45%)	1/23 (4.35%)	4/49 (8.16%)	1/48 (2.08%)	2/67 (2.99%)	3.50%	2.36%
	Slither	**22/47 (46.81%)**	**9/12 (75%)**	**7/58 (12.07%)**	**17/86 (19.77%)**	3/68 (4.41%)	**14.13%**	**29.05%**
	SmartCheck	16/53 (30.19%)	2/46 (4.35%)	5/57 (8.77%)	3/57 (5.26%)	**3/67 (4.48%)**	8.13%	11.09%
	Solhint	2/60 (3.33%)	2/54 (3.70%)	1/64 (1.56%)	3/59 (5.08%)	1/68 (1.47%)	1.96%	1.53%
	SolidityCheck	11/60 (18.33%)	1/48 (2.08%)	1/63 (1.59%)	1/51 (1.96%)	0/68 (0.00%)	2.63%	7.62%
	Vandal	3/60 (5.00%)	1/45 (2.22%)	1/60 (1.67%)	3/52 (5.77%)	1/68 (1.47%)	2.14%	2.01%
UEW	Ethainter	**10/51 (19.61%)**	1/36 (2.78%)	3/47 (6.38%)	5/46 (10.87%)	2/62 (3.23%)	5.86%	6.96%
	Mythril	2/60 (3.33%)	3/62 (4.84%)	**9/48 (18.75%)**	3/57 (5.26%)	**5/58 (8.62%)**	**8.49%**	6.23%
	Semgrep	0/0 (0.00%)	0/21 (0.00%)	0/35 (0.00%)	0/0 (0.00%)	0/5 (0.00%)	0.00%	0.00%
	Slither	3/65 (4.62%)	**7/58 (12.07%)**	2/66 (3.03%)	**12/64 (18.75%)**	0/68 (0.00%)	2.11%	**7.61%**
	Vandal	1/65 (1.54%)	4/83 (4.82%)	3/43 (6.98%)	4/71 (5.63%)	3/59 (5.08%)	4.80%	2.01%
	VeriSmart	0/67 (0.00%)	2/59 (3.39%)	1/67 (1.49%)	2/66 (3.03%)	0/68 (0.00%)	0.39%	1.61%
IOU	MadMax	**15/58 (25.86%)**	6/22 (27.27%)	12/43 (27.91%)	26/94 (27.66%)	**1/63 (1.59%)**	9.02%	11.47%
	Mythril	14/67 (20.90%)	21/57 (36.84%)	**32/66 (48.48%)**	9/61 (14.75%)	0/0 (0.00%)	8.87%	18.94%
	Semgrep	1/12 (8.33%)	**10/10 (100.00%)**	12/81 (14.81%)	**20/31 (64.52%)**	0/53 (0.00%)	**9.22%**	**43.03%**
	SmartCheck	10/54 (18.52%)	0/0 (0.00%)	0/0 (0.00%)	0/0 (0.00%)	0/0 (0.00%)	2.36%	8.28%
	VeriSmart	14/68 (20.59%)	15/68 (22.06%)	12/68 (17.65%)	7/67 (10.45%)	1/67 (1.49%)	6.51%	8.51%

[a] TP rate = True Positives / Total audited contracts (Percentage)
[b] The weighted total is the sum of each version's TP rate multiplied by its proportion of the total contract count in the dataset.
[c] σ denotes the standard deviation of the TP rate for a specific tool in versions 0.4.x to 0.8.x.
[d] For each type of vulnerability, the highest value among the tools in comparison is highlighted in **bold**.

[29] have reported that, in real-world contract datasets, only 1.98% and 0.015% of the vulnerable contracts reported by existing tools have actually been exploited, respectively.

> ***Finding 5:*** The manually confirmed TP rates of the tools remain below 30%, highlighting their limited accuracy in large-scale real-world smart contract analysis.

Adaptability of Tools according to True Positive Confirmation. As shown in Table 7, most tools show significant fluctuations in TP rates between different versions. For example, the tool with the highest TP rate for a specific vulnerability varies between different versions. In the following, we explore the adaptability of the tools by analyzing the variations in TP rates between different versions for each vulnerability.

REN As versions are upgraded, the changes in TP rates vary for each tool. Firstly, Semgrep shows the highest total TP rate (38.58%) and the highest standard deviation (22.93%). Specifically, although Semgrep achieves the highest TP rate in 0.8.x and the second highest in 0.6.x, its TP rate drops to 0 in all other versions. This is mainly due to its targeted detection of a specific reentrancy pattern [8] in ERC-721 token contracts, which is common in high versions. Secondly, Vandal has the second highest total TP rate (17.43%) and standard deviation 6.09%, which demonstrates effective performance and adaptability. Third, Slither and Mythril perform well in versions 0.4.x and 0.5.x, and 0.6.x and 0.7.x, respectively, but their performance significantly declines in version 0.8.x, because they not only generate false positives for common reentrancy protections (e.g., Checks-Effects-Interactions

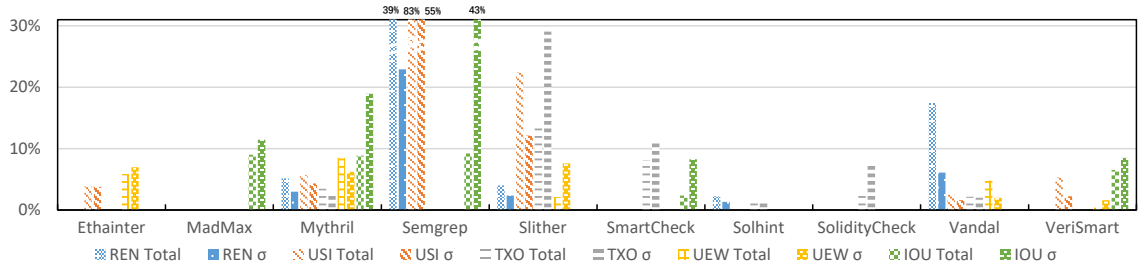

Fig. 9. Weighted total TP rate and standard deviation for each tool.

[53] and privileged users such as owner) in higher versions but also fail to detect new reentrancy patterns identified by Semgrep. Finally, Solhint incorrectly identified the built-in functions `transfer` and `send` as causes of reentrancy, resulting in nearly all results being false positives.

USI On the whole, the tools exhibit significantly higher TP rates in versions 0.4.x and 0.8.x compared to other versions. In detail, similar to REN, Semgrep achieves the highest total TP rate (83.36%) in 0.8.x and 0.4.x, but records a 0% TP rate across all other versions. It only reported a total of 4 vulnerabilities, indicating limited detection capability. In addition, Slither shows the second-highest total TP rate (22.39%) and standard deviation (12.10%). It has a significantly higher TP rate (12%) in version 0.7.x compared to other tools, but its TP rate is zero in versions 0.5.x and 0.6.x. Likewise, the TP rates for Ethainter, Mythril, and Vandal are almost zero in versions 0.5.x through 0.7.x as well. This suggests that contracts in these versions implemented specific protections for the self-destruct functions, which are overlooked by the tools. Moreover, the TP rate of VeriSmart is stable but its total TP rate (5.32%) is low, indicating limited effectiveness.

TXO In general, the tools exhibit lower TP rates in higher versions compared to lower versions. For example, Slither has the highest TP rate in versions 0.4.x to 0.7.x, but the TP rate declines significantly in version 0.8.x. SmartCheck and SolidityCheck have significantly higher TP rates in version 0.4.x compared to other versions. This is due to the fact that many contracts in the higher version consciously check both `tx.origin` and `msg.sender` permissions, such as `require(tx.origin == msg.sender)`, which tools incorrectly reported as dangerous. Moreover, although Mythril, Solhint, and Vandal have relatively small standard deviations (less than 3%), their total TP rates are below 4%, reflecting poor performance.

UEW The TP rates of the tools show significant differences across different versions. For example, Ethainter and Mythril have similar and high total TP rates (5% to 9%) and standard deviations (6% to 7%). However, each tool demonstrates its own strengths by exhibiting the highest TP rates in different versions. This is because the tools are mainly proficient at detecting some specific contract types in certain versions, such as Mythril detecting a total of seven AaveBasicProxyV2 contracts with UEW in version 0.6.x. Moreover, like REN, Vandal exhibits a stable TP rate but moderate effectiveness. Notably, despite having very small standard deviations (< 2%), Semgrep and VeriSmart both show very low TP rates (< 2%), indicating consistently poor performance.

IOU The TP rate for all tools dropped significantly in version 0.8.x (< 2%), because the default overflow checks for arithmetic operations [54] introduced in this version are overlooked by these tools. Additionally, from versions 0.4.x to 0.7.x, MadMax, Mythril, and VeriSmart maintain stable TP rates. However, Semgrep shows exceptionally high TP rates in versions 0.5.x (100%) and 0.7.x (64.52%), mainly because it reports a small number of vulnerabilities, almost exclusively in a specific type of contract. Furthermore, as mentioned in § 5.2, SmartCheck only reports vulnerabilities in version 0.4.x, highlighting its limitations.

Summary. Figure 9 shows the total TP rate and standard deviation for each tool. Overall, half of the tools (MadMax, Mythril, Semgrep, Slither, and SmartCheck) show significant fluctuations in TP rates across different versions ($\sigma \geq 10\%$). Among the remaining tools, four (Ethainter, Solhint, SolidityCheck, and VeriSmart) exhibit low total TP rates ($\leq 10\%$) despite having small standard deviations, indicating poor effectiveness. The remaining tool (Vandal) achieves moderate effectiveness (Total TP rate between 0% and 20%) and good stability ($\sigma \leq 10\%$) when detecting REN and IOU vulnerabilities, respectively.

The underlying causes of significant fluctuations in tool TP rates with version upgrades can be attributed to two key factors: (1) A decrease in TP rate stems from enhanced compiler protections (e.g., IOU in 0.8.x) and developers' proficient use of various defense strategies, which the tool may overlook. (2) An increase in the TP rate occurs when tools' rules are specifically designed to detect certain contract types (e.g., Semgrep for ERC-721 token contracts), leading to highly effective results in specific versions.

> ***Finding 6:*** Analysis of TP rate variations across compiler versions reveals that half of the tools exhibit poor adaptability, primarily due to inadequate handling of compiler-level and developer-enforced security measures.

5.4 Effectiveness Evaluation on Manually Confirmed Results

After manually verifying the true positive and false positive samples reported by each tool, we aggregate them to construct a derived ground-truth dataset. This dataset serves as the foundation for a controlled evaluation that incorporates both vulnerable and non-vulnerable smart contracts, enabling a more comprehensive evaluation of each tool's detection effectiveness and supporting the facilitating analysis of tool adaptability across different versions.

Metrics. We measure the number of true positives (TPs), false positives (FPs), false negatives (FNs), and true negatives (TNs) for each tool, and compute the following standard evaluation metrics:

$$Precision = \frac{TP}{TP + FP},\ Recall = \frac{TP}{TP + FN},\ F1\ Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{2}$$

Evaluation Results. As shown in Table 8, columns 3 to 7 report the evaluation results of each tool across different vulnerability types and contract versions, while the last column summarizes the overall effectiveness. Each cell includes the counts of TPs, FPs, FNs, and TNs, along with the corresponding precision, recall, and F1 score.

Overall, based on the F1 score, Slither demonstrates the best performance, achieving the highest scores on the USI, TXO, and UEW vulnerabilities. Semgrep performs best on REN, while VeriSmart attains the highest F1 score on IOU. In terms of recall, Slither again leads on USI and UEW, while Vandal, Solhint, and VeriSmart achieve the best recall on REN, TXO, and IOU, respectively. Regarding precision, Semgrep attains the highest values on REN and USI, Slither on TXO, Ethainter on UEW, and Mythril on IOU.

Adaptability of Tools according to Effectiveness Evaluation. We analyze the adaptability of each tool by examining how its precision, recall, and the combined metric F1 score vary across different versions. A tool is considered to have good adaptability if its evaluation metrics remain stable or improve as the version evolves. In contrast, a significant decline in these metrics indicates poor adaptability. In the following, we present the adaptability trends of different tools for each type of vulnerability.

For the REN vulnerability, most tools (e.g., Mythril, Slither, and Solhint) show a trend of slightly increasing precision but noticeably declining recall across newer versions, suggesting a growing tendency to under-report vulnerabilities while maintaining correctness when they do report. Semgrep performs well only on specific versions

Table 8. The manually confirmed effectiveness of the tools across different versions.

	Tool	0.4.x[a]	0.5.x[a]	0.6.x[a]	0.7.x[a]	0.8.x[a]	Total[a]
REN	Mythril	7 | 80 | 5 | 202 **8%** | 58% | **14%**	4 | 47 | 10 | 225 **8%** | 29% | 12%	13 | 63 | 16 | 222 **17%** | 45% | **25%**	6 | 50 | 12 | 239 **11%** | 33% | **16%**	11 | 77 | 46 | 205 13% | 19% | 15%	41 | 317 | 89 | 1093 11% | 32% | 17%
	Semgrep	0 | 31 | 12 | 251 0% | 0% | #	3 | 79 | 11 | 193 4% | 21% | 6%	9 | 78 | 20 | 207 10% | 31% | 16%	1 | 95 | 17 | 194 1% | 6% | 2%	51 | 87 | 6 | 195 **37%** | 89% | **52%**	64 | 370 | 66 | 1040 **15%** | 49% | **23%**
	Slither	12 | 184 | 0 | 98 6% | **100%** | 12%	12 | 163 | 2 | 109 7% | 86% | **13%**	18 | 230 | 11 | 55 7% | 62% | 13%	12 | 210 | 6 | 79 5% | 67% | 10%	37 | 220 | 20 | 62 14% | 65% | 24%	91 | 1007 | 39 | 403 8% | 70% | 15%
	Solhint	1 | 122 | 11 | 160 1% | 8% | 1%	1 | 61 | 13 | 211 2% | 7% | 3%	2 | 80 | 27 | 205 2% | 7% | 4%	3 | 79 | 15 | 210 4% | 17% | 6%	2 | 70 | 55 | 212 3% | 4% | 3%	9 | 412 | 121 | 998 2% | 7% | 3%
	Vandal	11 | 263 | 1 | 19 | 4% | 92% | 8%	14 | 231 | 0 | 41 | 6% | **100%** | 11%	28 | 267 | 1 | 18 | 9% | **97%** | 17%	18 | 267 | 0 | 22 | 6% | **100%** | 12%	55 | 267 | 2 | 15 | 17% | **96%** | 29%	126 | 1295 | 4 | 115 | 9% | **97%** | 16%
USI	Ethainter	7 | 95 | 2 | 111 | 7% | 78% | 13%	0 | 24 | 0 | 20 | 0% | # | #	0 | 20 | 1 | 52 | 0% | 0% | #	2 | 87 | 1 | 72 | 2% | 67% | 4%	10 | 115 | 14 | 120 | 8% | 42% | 13%	19 | 341 | 18 | 375 | 5% | 51% | 10%
	Mythril	6 | 62 | 3 | 144 | 9% | 67% | 16%	0 | 4 | 0 | 40 | 0% | # | #	1 | 7 | 0 | 65 | **13%** | **100%** | **22%**	2 | 16 | 1 | 143 | **11%** | 67% | 19%	8 | 70 | 16 | 165 | 10% | 33% | 16%	17 | 159 | 20 | 557 | 10% | 46% | 16%
	Semgrep	1 | 0 | 8 | 206 | **100%** | 11% | **20%**	0 | 0 | 0 | 44 | # | # | #	0 | 0 | 1 | 72 | # | 0% | #	0 | 0 | 3 | 159 | # | 0% | #	3 | 1 | 21 | 234 | **75%** | 13% | 21%	4 | 1 | 33 | 715 | **80%** | 11% | 19%
	Slither	8 | 86 | 1 | 120 | 9% | 89% | 16%	0 | 16 | 0 | 28 | 0% | # | #	1 | 8 | 0 | 64 | 11% | **100%** | 20%	3 | 25 | 0 | 134 | 11% | **100%** | **19%**	23 | 57 | 1 | 178 | 29% | **96%** | **44%**	35 | 192 | 2 | 524 | 15% | **95%** | **27%**
	Vandal	5 | 81 | 4 | 125 | 6% | 56% | 11%	0 | 7 | 0 | 37 | 0% | # | #	0 | 40 | 1 | 32 | 0% | 0% | #	0 | 43 | 3 | 116 | 0% | 0% | #	8 | 101 | 16 | 134 | 7% | 33% | 12%	13 | 272 | 24 | 444 | 5% | 35% | 8%
	VeriSmart	9 | 158 | 0 | 48 | 5% | **100%** | 10%	0 | 31 | 0 | 13 | 0% | #| #	1 | 30 | 0 | 42 | 3% | **100%** | 6%	3 | 95 | 0 | 64 | 3% | **100%** | 6%	6 | 59 | 18 | 176 | 9% | 25% | 13%	19 | 373 | 18 | 343 | 5% | 51% | 9%
TXO	Mythril	26 | 157 | 12 | 61 | 14% | 68% | 24%	5 | 22 | 7 | 126 | 19% | 42% | 26%	5 | 75 | 13 | 231 | 6% | 28% | 10%	3 | 47 | 16 | 177 | 6% | 16% | 9%	4 | 120 | 6 | 276 | 3% | 40% | 6%	43 | 421 | 54 | 871 | 9% | 44% | 15%
	Slither	24 | 25 | 14 | 193 | **49%** | 63% | **55%**	9 | 10 | 3 | 138 | **47%** | 75% | **58%**	14 | 135 | 4 | 171 | 9% | 78% | 17%	18 | 74 | 1 | 150 | **20%** | 95% | **32%**	9 | 200 | 1 | 196 | 4% | 90% | 8%	74 | 444 | 23 | 848 | **14%** | 76% | **24%**
	SmartCheck	38 | 92 | 0 | 126 | 29% | **100%** | 45%	5 | 71 | 7 | 77 | 7% | 42% | 11%	14 | 127 | 4 | 179 | **10%** | 78% | **18%**	14 | 153 | 5 | 71 | 8% | 74% | 15%	7 | 136 | 3 | 260 | **5%** | 70% | **9%**	78 | 579 | 19 | 713 | 12% | 80% | 21%
	Solhint	31 | 197 | 7 | 21 | 14% | 82% | 23%	12 | 147 | 0 | 1 | 8% | **100%** | 14%	16 | 292 | 2 | 14 | 5% | **89%** | 10%	19 | 211 | 0 | 13 | 8% | **100%** | 15%	10 | 388 | 0 | 8 | 3% | **100%** | 5%	88 | 1235 | 9 | 57 | 7% | **91%** | 12%
	SolidityCheck	37 | 191 | 1 | 27 | 16% | 97% | 28%	12 | 100 | 0 | 48 | 11% | **100%** | 19%	12 | 267 | 6 | 39 | 4% | 67% | 8%	15 | 140 | 4 | 84 | 10% | 79% | 17%	6 | 369 | 4 | 27 | 2% | 60% | 3%	82 | 1067 | 15 | 225 | 7% | 85% | 13%
	Vandal	24 | 188 | 14 | 30 | 11% | 63% | 19%	4 | 79 | 8 | 69 | 5% | 33% | 8%	13 | 174 | 5 | 132 | 7% | 72% | 13%	15 | 121 | 4 | 103 | 11% | 79% | 19%	9 | 230 | 1 | 166 | 4% | 90% | 7%	65 | 792 | 32 | 500 | 8% | 67% | 14%
UEW	Ethainter	13 | 49 | 3 | 238 | **21%** | **81%** | **33%**	1 | 35 | 13 | 233 | 3% | 7% | 4%	4 | 53 | 12 | 230 | 7% | 25% | 11%	11 | 63 | 13 | 205 | 15% | 46% | 22%	6 | 63 | 4 | 246 | **9%** | 60% | **15%**	35 | 263 | 45 | 1152 | **12%** | 44% | 19%
	Mythril	4 | 68 | 12 | 219 | 6% | 25% | 9%	3 | 60 | 11 | 208 | 5% | 21% | 8%	10 | 40 | 6 | 243 | **20%** | 63% | **30%**	4 | 54 | 20 | 214 | 7% | 17% | 10%	5 | 56 | 5 | 253 | 8% | 50% | 14%	26 | 278 | 54 | 1137 | 9% | 33% | 14%
	Semgrep	0 | 0 | 16 | 287 | # | 0% | #	0 | 21 | 14 | 247 | 0% | 0% | #	0 | 35 | 16 | 248 | 0% | 0% | #	0 | 0 | 24 | 268 | # | 0% | #	0 | 5 | 10 | 304 | 0% | 0% | #	0 | 61 | 80 | 1354 | 0% | 0% | #
	Slither	12 | 106 | 4 | 181 | 10% | 75% | 18%	14 | 102 | 0 | 166 | **12%** | **100%** | **22%**	13 | 100 | 3 | 183 | 12% | **81%** | 20%	20 | 106 | 4 | 162 | **16%** | **83%** | **27%**	7 | 129 | 3 | 180 | 5% | **70%** | 10%	66 | 543 | 14 | 872 | 11% | **83%** | **19%**
	Vandal	6 | 91 | 10 | 196 | 6% | 38% | 11%	4 | 80 | 10 | 188 | 5% | 29% | 8%	5 | 43 | 11 | 240 | 10% | 31% | 16%	5 | 68 | 19 | 200 | 7% | 21% | 10%	3 | 58 | 7 | 251 | 5% | 30% | 8%	23 | 340 | 57 | 1075 | 6% | 29% | 10%
	VeriSmart	13 | 186 | 3 | 101 | 7% | **81%** | 12%	12 | 110 | 2 | 158 | 10% | 86% | 18%	7 | 156 | 9 | 127 | 4% | 44% | 8%	8 | 137 | 16 | 131 | 6% | 33% | 9%	5 | 97 | 5 | 212 | 5% | 50% | 9%	45 | 686 | 35 | 729 | 6% | 56% | 11%
IOU	MadMax	18 | 44 | 36 | 157 | **29%** | 33% | 31%	6 | 16 | 45 | 90 | 27% | 12% | 16%	12 | 31 | 56 | 160 | 28% | 18% | 22%	26 | 69 | 36 | 120 | 27% | 42% | 33%	1 | 62 | 1 | 119 | **2%** | **50%** | **3%**	63 | 222 | 174 | 646 | 22% | 27% | 24%
	Mythril	25 | 86 | 29 | 115 | 23% | 46% | 30%	21 | 38 | 30 | 68 | 36% | 41% | 38%	35 | 42 | 33 | 149 | **45%** | 51% | **48%**	11 | 55 | 51 | 134 | 17% | 18% | 17%	0 | 0 | 2 | 181 | # | 0% | #	92 | 221 | 145 | 647 | **29%** | 39% | 33%
	Semgrep	1 | 11 | 53 | 190 | 8% | 2% | 3%	10 | 0 | 41 | 106 | **100%** | 20% | 33%	12 | 69 | 56 | 122 | 15% | 18% | 16%	20 | 11 | 42 | 178 | **65%** | 32% | **43%**	0 | 53 | 2 | 128 | 0% | 0% | #	43 | 144 | 194 | 724 | 23% | 18% | 20%
	SmartCheck	11 | 44 | 43 | 157 | 20% | 20% | 20%	0 | 0 | 51 | 106 | # | 0% | #	0 | 0 | 68 | 191 | # | 0% | #	0 | 0 | 62 | 189 | # | 0% | #	0 | 0 | 2 | 181 | # | 0% | #	11 | 44 | 226 | 824 | 20% | 5% | 8%
	VeriSmart	49 | 140 | 5 | 61 | 26% | **91%** | **40%**	38 | 87 | 13 | 19 | 30% | **75%** | **43%**	46 | 90 | 22 | 101 | 34% | **68%** | 45%	29 | 112 | 33 | 77 | 21% | **47%** | 29%	1 | 68 | 1 | 113 | 1% | **50%** | 3%	163 | 497 | 74 | 371 | 25% | **69%** | **36%**

[a] Each cell reports the following metrics: TPs | FPs | FNs | TNs | Precision | Recall | F1 Score

[b] The symbol "#" indicates a division-by-zero error.

[c] For each type of vulnerability, the highest value among the tools in comparison is highlighted in **bold**.

(notably 0.6.x and 0.8.x), indicating limited adaptability. In contrast, Vandal exhibits consistent improvement across versions, with its F1 score rising from 8% to 29%, reflecting enhanced detection capability on newer Solidity versions.

For the USI vulnerability, most tools exhibit noticeable effectiveness shifts on version 0.8.x. Tools such as Ethainter, Mythril, Vandal, and VeriSmart show a slight increase in precision but a marked decline in recall, suggesting a

growing tendency to miss vulnerabilities while preserving accuracy in detected cases. Semgrep again performs well only on specific versions (0.5.x and 0.8.x), indicating limited generalizability. In contrast, Slither maintains stable effectiveness across versions, demonstrating strong adaptability.

For the TXO vulnerability, all tools exhibit a noticeable decrease in precision, while half of the tools (Mythril, SmartCheck, and SolidityCheck) also show a significant decline in recall. The remaining tools display a slight increase in recall, but not enough to offset the overall degradation in detection effectiveness. These trends suggest that the fixed and simplified TXO patterns in newer versions tend to introduce more false positives for most tools.

For the UEW vulnerability, some tools maintain stable effectiveness across versions, while others exhibit a noticeable decline. Specifically, Mythril, Vandal and Slither maintain relatively stable effectiveness, with slight improvements. In contrast, Ethainter and VeriSmart show a clear decline in both precision and recall, indicating reduced reliability in detecting UEW vulnerabilities. Semgrep performs poorly, reporting no true positives across all versions.

For the IOU vulnerability, the effectiveness of all tools experiences a sharp decline in version 0.8.x. Except for MadMax and VeriSmart, whose recall remains around 50%, the effectiveness metrics of all other tools fall to nearly zero, indicating a near-complete failure to detect IOU vulnerabilities in this version.

Summary. Overall, only a few tools demonstrate good adaptability—Vandal on REN and UEW, Slither on USI and UEW, and Mythril on UEW. For most other tools and vulnerability types, adaptability remains limited. In terms of precision, slight improvements are observed on REN and USI, whereas a decline is evident in all other cases. Regarding recall, all tools exhibit a consistent downward trend. Notably, tool effectiveness shows the most pronounced variation in version 0.8.x (e.g., IOU), indicating substantial changes in detection behavior.

> ***Finding 7:*** Effectiveness evaluation results indicate that Vandal, Slither, and Mythril exhibit good adaptability on specific vulnerabilities, whereas most other tools experience notable degradation, particularly in version 0.8.x.

6 RQ3: Consistency of Static Tools

In this section, we evaluate how the consistency of detection results for tools changes with contract upgrades, and conduct a comprehensive analysis of consistency across various combinations of tools to identify the optimal configuration.

6.1 Experimental Setup.

In the experiment, we evaluate the consistency of the ten static tools that are compatible with all versions of the smart contract shown in § 4.2 on the SESC dataset. Since the detection results of the ten static tools on the SESC dataset, focusing on five high-risk vulnerabilities, are obtained in § 5.1 of RQ2, we proceed directly with the statistical analysis.

6.2 Overlap Analysis

We evaluate the consistency of the tools by analyzing the overlap in their results. In detail, for each vulnerability, the addresses of vulnerable contracts reported by each tool for a specific version form a set. Overlap in detection results is measured by calculating the similarity between sets. Thus, we compute the similarity between these sets using the Jaccard Similarity Coefficient [32].

Given the numerous combinations of tools, we categorize these combinations according to the number of tools they contain, and calculate the average Jaccard Similarity Coefficient for each group. The definition is as follows.

Definition 6.1 (Average Jaccard Similarity Coefficient). For each vulnerability type and version, given a tool set $T = (t_1, t_2, ..., t_n)$ and their result sets $\mathbb{V} = (V_{t_1}, V_{t_2}, ..., V_{t_n})$, where V_{t_n} is a set that contains vulnerable contract addresses

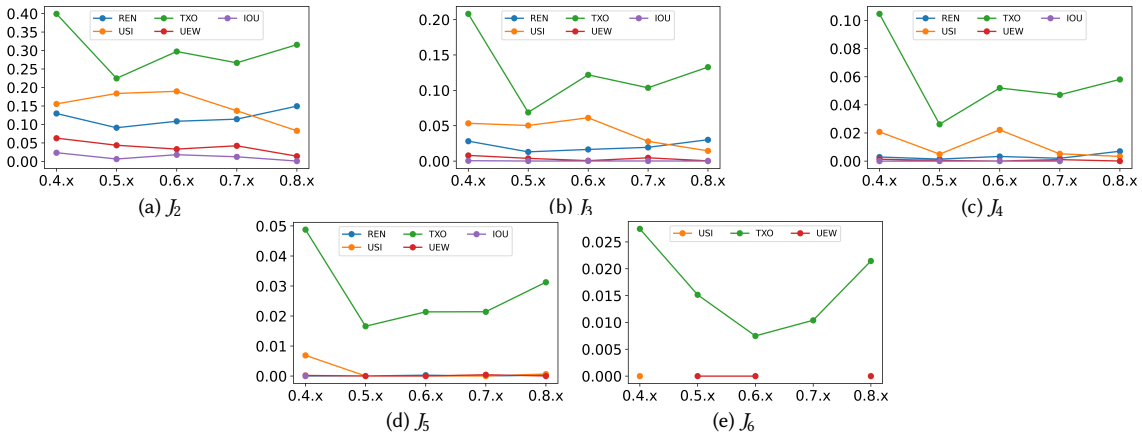

Fig. 10. The average Jaccard Similarity Coefficient of tools.

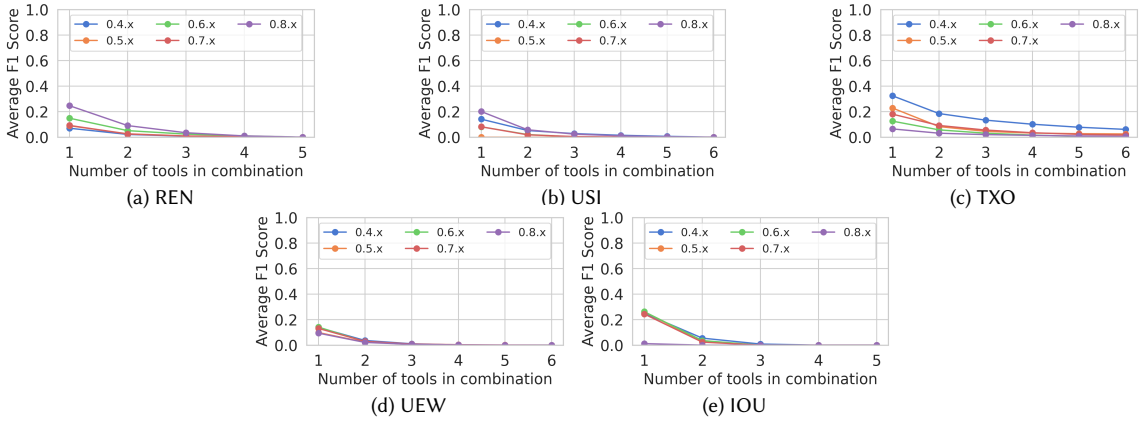

Fig. 11. Average F1 score of different combinations of tools.

reported by tool t_n. The average Jaccard Similarity Coefficient of tools can be calculated as shown in Equation 3, where J_2, J_3, ..., J_n represent the average Jaccard Similarity Coefficient for pairs, triplets, and n-tuples of tools, respectively.

$$\mathrm{J_2} = \frac{1}{\binom{n}{2}} \sum_{i<j} \frac{|V_{t_i} \cap V_{t_j}|}{|V_{t_i} \cup V_{t_j}|} \quad \mathrm{J_3} = \frac{1}{\binom{n}{3}} \sum_{i<j<k} \frac{|V_{t_i} \cap V_{t_j} \cap V_{t_k}|}{|V_{t_i} \cup V_{t_j} \cup V_{t_k}|} \quad \ldots \quad \mathrm{J_n} = \frac{1}{\binom{n}{n}} \sum \frac{|V_{t_1} \cap V_{t_2} \ldots \cap V_{t_n}|}{|V_{t_1} \cup V_{t_2} \ldots \cup V_{t_n}|} \tag{3}$$

Results. Figure 10 shows the average Jaccard Similarity Coefficient for tool combinations of sizes 2 to 6, across each vulnerability and version, as each type of vulnerability can be detected by up to 6 tools out of 10. Note that REN and USI are supported by only five tools, and therefore do not have J_6 values. The average Jaccard Similarity Coefficients from J_2 to J_6 exhibit a multiplicative decrease, because the overlap diminishes with the increasing number of tools. Overall, the tools show the greatest consistency in detecting TXO, followed by USI and REN, while consistency is lowest for UEW and IOU vulnerabilities.

With version upgrades: (1) For REN, J2 to J5 slightly decline from version 0.4.x and then gradually recover. (2) For USI, J2 to J4 exhibit a significant decline starting from version 0.6.x, J5 drops from 0.4.x to near 0 and remains constant, while J6 shows overlap only in version 0.4.x. (3) For TXO, J2 to J6 show a significant drop from version 0.4.x, followed by a partial recovery. (4) For UEW, J2 begins to decrease from version 0.4.x, with subsequent versions showing lower

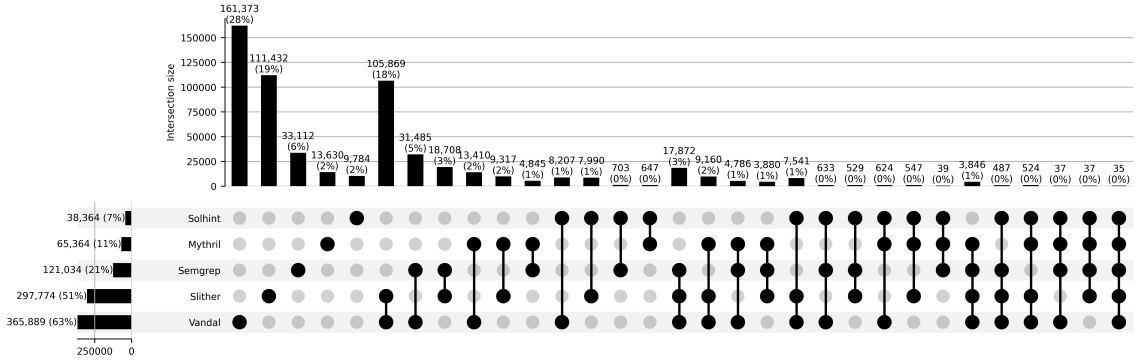

Fig. 12. The UpSet diagram of vulnerable contracts with REN detected by the tools.

values, and J3 to J6 remain stable at low values. (5) For IOU, there is minimal variation in J2 to J5, but J2 in version 0.8.x is notably the lowest.

In general, the tools demonstrate notable variations in consistency across different versions. Specifically, for most vulnerabilities, except for REN, the tools demonstrate lower consistency in higher versions than in lower versions. For example, the consistency of tools in detecting TXO, UEW, and IOU on version 0.4.x is higher than in other versions, and the tools exhibited lower consistency in detecting USI on versions 0.7.x and 0.8.x compared to the lower versions. Moreover, for REN, while the tools demonstrate high consistency on version 0.8.x, all other versions still fall short compared to version 0.4.x.

> ***Finding 8:*** With each version upgrade, the consistency of static tools has exhibited a certain degree of decline for most vulnerabilities and versions.

Manual Confirmation. In § 5.4, we evaluate the effectiveness of individual tools across different versions and vulnerability types. Considering the significant inconsistency among tools, we further measure the impact of combining multiple tools on detection effectiveness. We hypothesize that the results consistently agreed upon by multiple tools are more likely to be correct, whereas conflicting results are more likely to be false. Therefore, integrating more tools should, in principle, improve overall effectiveness. However, as shown in Figure 11, the line chart illustrates that the average F1 score decreases as more tools are combined, compared to using a single tool. This observation contradicts the hypothesis, suggesting that the tools should be regarded as complementary rather than conflicting in their results.

> ***Finding 9:*** The low consistency among tools suggests that they are complementary rather than conflicting.

Tool Combinations. To identify the optimal tool combination, we present the overlap across all tool combinations, using the highest-risk REN vulnerability as a representative example (details for other vulnerabilities can be found in [1]). As shown in Figure 12, the UpSet diagram visualizes the overlap in vulnerable contracts detected by the tools. At the bottom, tool combinations are depicted as a matrix of dots, where each dot represents a tool involved in a particular combination. Above the matrix, the intersection size bar chart indicates the number of vulnerable contracts corresponding to each tool combination. On the left side, a vertical bar chart shows the total number of vulnerable contracts belonging to each tool in the top intersection size bar chart.

Overall, Vandal not only identifies the highest number of vulnerable contracts, but also covers nearly all detected by other tools, demonstrating a remarkably comprehensive detection capability. Specifically, the first column of the dot matrix represents the highest number of vulnerable contracts detected by Vandal. The second and sixth columns show

Table 9. Comparison of approaches and detection rules used by tools.

Tools	Approach	Level	Detection Rules[a]
Ethainter	Symbolic Analysis	Bytecode	**#USI:** Accessible selfdestruct vulnerability is more commonly encountered in combination with a tainted guard. [9]
MadMax	IR Matching	Bytecode	**#IOU:** Loop overflows are conservatively asserted to be likely if the induction variable is cast to a short integer or ideally one byte. The loop has to be dynamically bound to be vulnerable, i.e., the number of iterations is determined by some run-time value. [28]
Mythril	Symbolic Analysis	Bytecode	**#REN:** If an external call to an untrusted address is detected, analyze the control flow graph for possible state changes that occur after the call returns. [3]
Semgrep	Pattern Matching	Source Code	**#REN:** (1) Function `borrowFresh()` in Compound performs state update after `doTransferOut()`. (2) ERC677 with `callAfterTransfer()` or ERC777 with `tokensReceived()` or ERC721 with `onERC721Received()`. (3) `getRate()` or `getPoolTokens()` or `get_virtual_price()` call on a Balancer pool is not protected from the read-only reentrancy. [2]
Slither	IR Matching	Source Code	**#REN:** A state variable is changed only after an external call (`function call()`). [23]
SmartCheck	IR Matching	Source Code	**#TXO:** The pattern detects the environmental variable `tx.origin`. [57]
Solhint	Pattern Matching	Source Code	**#REN:** Avoid state changes after transfer. [4]
SolidityCheck	IR Matching	Source Code	**#TXO:** Using `tx.origin` for authentication. [64]
Vandal	IR Matching	Bytecode	**#REN:** A call is flagged as reentrant if it forwards sufficient gas and is not protected by a mutex. [10]
VeriSmart	Symbolic Analysis	Source Code	**#USI**: Given a statement that deactivates contracts, we report a suicidal vulnerability if the statement can be executed by untrusted users. [50]

[a] The complete detection rules for the five types of vulnerabilities analyzed by the tools can be found in [1].

the contracts detected by Slither (19%) and the overlap between Vandal and Slither (18%), respectively. This overlap accounts for nearly all the results identified by Slither (18%/19%). A similar pattern is observed for other tools, such as Semgrep (5%/6%), Mythril (2%/2%), and Solhint (1%/2%). Therefore, Vandal alone can be regarded as the optimal tool combination for REN.

Furthermore, for other vulnerabilities, we identify the following optimal tool combinations: (1) For USI, since Ethainter and Mythril cover more than three-quarters of the results from Slither and Semgrep, the optimal tool combination is Ethainter, Mythril, Vandal and VeriSmart. (2) For TXO, Solhint covers almost entirely the results of other tools, making it the best tool. (3) For UEW, there is very little overlap among the results of tools, so we recommend considering all available tools. (4) For IOU, since VeriSmart covers more than half of the results from Mythril and SmartCheck, the optimal combination is VeriSmart, MadMax, and Semgrep.

> ***Finding 10:*** For each vulnerability type, an optimal combination of tools can be identified to achieve more comprehensive vulnerability coverage while reducing execution cost, such as Vandal for REN.

Reason Analysis. To further analyze the relationship between tool design and detection consistency, Table 9 presents approaches and detection rules of each tool, taking a vulnerability as an example. In detail, for REN detection, Slither and Vandal rely on the same intermediate representation matching approach, resulting in identical detection counts and highly similar results. Mythril, which employs symbolic execution, is constrained by path explosion, leading to fewer detected vulnerabilities. However, its results largely overlap with those of Slither and Vandal. Semgrep focuses on specific patterns derived from real-world DeFi exploits, offering limited detection capabilities but high similarity with Vandal. Solhint, in contrast, applies general rules but incorrectly targets the `transfer` function, leading to weaker alignment with other tools. For USI detection, Ethainter and VeriSmart both utilize symbolic analysis. However, VeriSmart operates at the source code level, extracting richer semantic information, resulting in slightly more detections but fewer overlapping findings with Ethainter. For TXO detection, SolidityCheck and SmartCheck share identical detection rules. However, SolidityCheck employs regex-based pattern matching, whereas SmartCheck leverages a customized intermediate representation. As a result, SolidityCheck has broader coverage and

```
1  function _safeMint(address to, uint256 quantity, bytes memory _data) internal {
2      uint256 startTokenId = currentIndex;
3      require(to != address(0), "ERC721A: mint to the zero address");
4      require(!_exists(startTokenId), "ERC721A: token already minted");
5      require(quantity > 0, "ERC721A: quantity must be greater 0");
6      _beforeTokenTransfers(address(0), to, startTokenId, quantity);
7      AddressData memory addressData = _addressData[to];
8      _addressData[to] = AddressData(
9          addressData.balance + uint128(quantity),
10         addressData.numberMinted + uint128(quantity));
11     _ownerships[startTokenId] = TokenOwnership(to, uint64(block.timestamp));
12     uint256 updatedIndex = startTokenId;
13     for (uint256 i = 0; i < quantity; i++) {
14         emit Transfer(address(0), to, updatedIndex);
15         require(
16             _checkOnERC721Received(address(0), to, updatedIndex, _data),
17             "ERC721A: transfer to non ERC721Receiver implementer"); // REN
18         updatedIndex++;
19     }
20     currentIndex = updatedIndex;
21     _afterTokenTransfers(address(0), to, startTokenId, quantity);
22 }
```

Fig. 13. A typical example of reentrancy vulnerability in version 0.8.x from address 0x83B06d09B99AD2641Dd9b1132E8Ce8809b623433.

subsumes most of SmartCheck 's results. For IOU detection, MadMax primarily targets loop overflow vulnerabilities. While its results exhibit limited overlap with other tools, a notable portion aligns with VeriSmart.

> ***Finding 11:*** Tools with similar analysis approaches and code levels tend to exhibit higher consistency, but differences in detection rules still lead to noticeable variation.

7 Discussion

In this section, we discuss the increase of bugs in newer versions, provide practical recommendations based on our findings from the previous sections, and outline potential threats to the validity of our results.

7.1 More Bugs in Newer Versions?

In RQ2, Table 4 shows that the detection rate of REN vulnerabilities increases in newer Solidity versions, and Table 7 confirms that many of these are true positives based on manual confirmation. As REN is a well-known vulnerability, its continued prevalence is counterintuitive and raises the question of why this occurs. We discuss potential reasons from two perspectives as follows. First, from the perspective of manual validation experience, we identify a typical REN vulnerability that frequently occurs in higher versions. Its prevalence is largely due to the widespread reuse of similar contract patterns, which leads to recurring vulnerabilities. Second, from a statistical perspective, we analyze the distribution of source code across different versions and observe significant differences. This suggests that applying security knowledge from earlier versions to newer ones requires substantial adaptation. While straightforward issues may be avoided, complex scenarios continue to introduce substantial risks.

Typical Case Analysis. Figure 13 presents a representative REN vulnerability frequently observed in higher versions during our manual confirmation process. This case has been widely discussed in the community (e.g., detailed in the blog [8]). Specifically, line 16 invokes `_checkOnERC721Received`, which can perform a reentrant call. However, the storage state variable `currentIndex` is updated later at line 20, creating a security risk. This function is widely reused in ERC-721

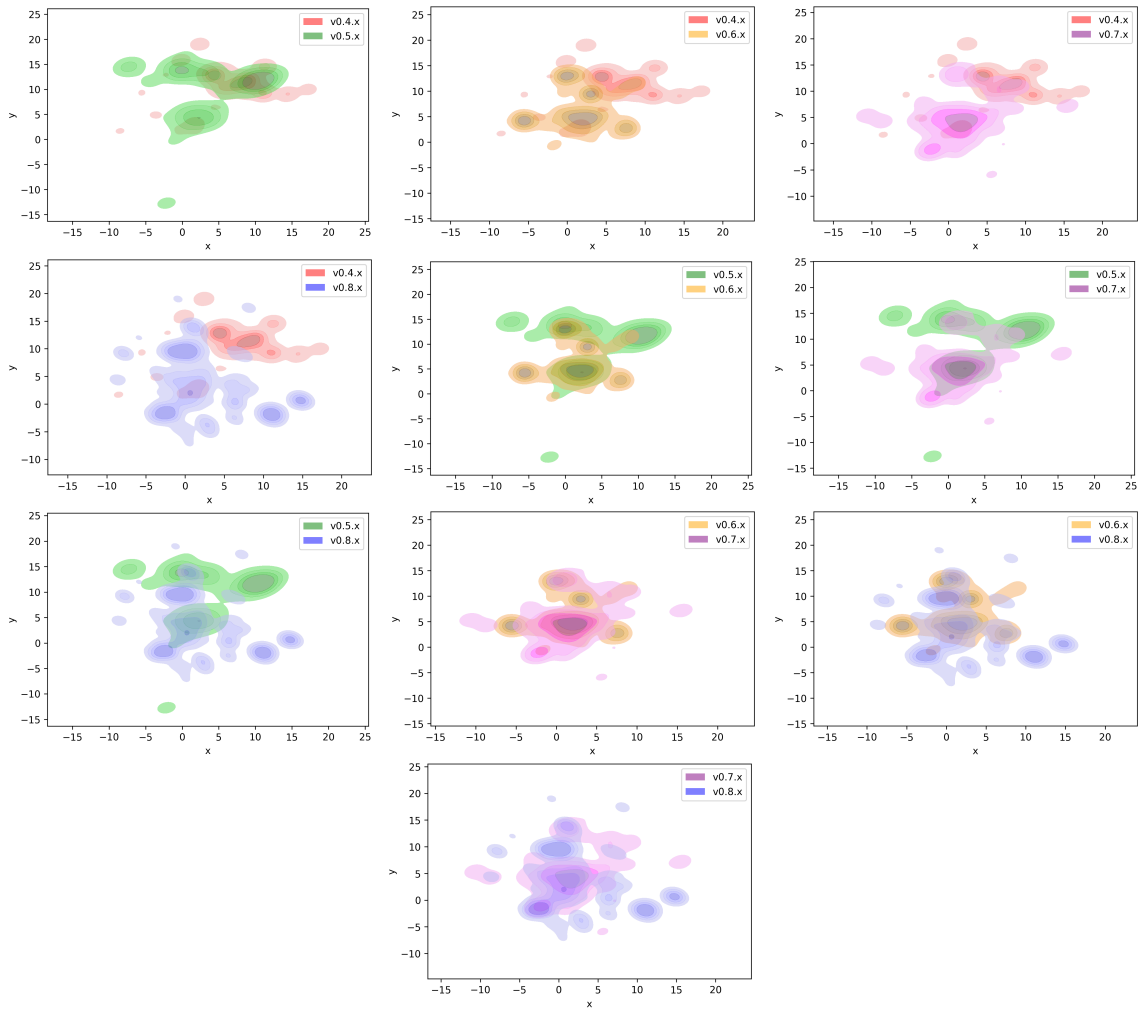

Fig. 14. KDE contour plots of smart contract distributions across pairwise comparisons of Solidity versions.

token contracts through copy-pasting and often assumed to be secure due to the misleading function name `_safeMint`, which lowers developers' vigilance. Moreover, the complexity of the context—such as numerous internal function calls and variable propagations—increases the difficulty of recognizing the reentrancy risk.

Contract Distribution Drift. To analyze the distributional characteristics of contracts across versions, we perform clustering analysis on the contract source code. Specifically, for each contract, we remove comments to obtain the pure code and embed it using OpenAI's `text-embedding-3-large` model [41], resulting in 3,072-dimensional semantic vectors. Due to the model's input length constraint (8,192 tokens), longer source files are split and the resulting segment embeddings are aggregated using mean pooling. Subsequently, we apply the HDBSCAN [11] algorithm for clustering, as it can identify clusters with varying densities and does not require the number of clusters to be specified in advance. Then, we use UMAP [39] to reduce the high-dimensional embeddings to two dimensions for visualization. UMAP effectively preserves both local and global structure in the reduced two-dimensional space.

Table 10. Checklist for developing cross-version compatible static tools.

Component	Checklist	Reason
Internal Analysis	☐ Ensure compatibility with semantic and syntactic changes introduced in each Solidity minor version. ☐ Handle newly introduced, modified, or deprecated EVM opcodes across Ethereum hard forks.	C1, C2 C4, C5
External Dependencies	☐ Keep all external packages (e.g., `solidity-parser` and `solcx`) maintained and updated. ☐ Align with the structure of Solidity compiler outputs, including ASTs and source mappings.	C2 C3
Performance	☐ Evaluate on smart contract projects with large codebases (e.g., over 1K LOC). ☐ Test across large-scale multi-version contract datasets.	C6 C7

As shown in Figure 14, we present kernel density estimation (KDE) contour plots to illustrate the density distribution of contracts across pairwise comparisons of Solidity versions. In these plots, the contour lines represent regions with a higher density of contracts that share similar semantic characteristics. The closer the contour lines are to each other, the steeper the density gradient, indicating a higher concentration of contracts in that region. The results reveal significant distributional shifts across versions: contracts in versions 0.4.x and 0.5.x differ substantially from those in version 0.8.x, while versions 0.6.x, 0.7.x, and 0.8.x exhibit considerable partial overlap. This suggests a notable semantic drift in contract development practices over time. Therefore, as reentrancy-prone code patterns evolve, it may become difficult for developers to transfer security knowledge from earlier versions to newer contexts, resulting in the continued presence of well-known vulnerabilities like REN in newer-version contracts.

7.2 Practical Recommendations

Based on our empirical findings, we provide targeted practical recommendations to guide future research efforts.

- **Select a compatible base tool for extension.** Given the trend of extending existing tools to support emerging vulnerability types (e.g., eTainter, AChecker), we recommend choosing a base tool that already supports the target Solidity versions to avoid losing core analysis capabilities. Based on the reasons that hinder tool compatibility (C1–C7) as identified in RQ1, we provide a practical checklist in Table 10 to support developers in building cross-version compatible static analysis tools. The checklist organizes key considerations into three components: (1) Internal analysis: adapt to semantic, syntactic, and EVM opcode changes; (2) External dependencies: ensure alignment with compiler outputs and keep dependency packages updated; (3) Performance: evaluate on large smart-contract projects and large-scale multi-version datasets.
- **Refine detection rules with version-aware semantics.** As shown in RQ2, many tools produce excessive false positives due to overlooking compiler-level and developer-level defense mechanisms, such as built-in overflow checks introduced in Solidity 0.8.x [54] and the widely adopted Checks-Effects-Interactions pattern for reentrancy prevention [27, 30]. These observations should serve as important lessons for future rule development, which must take defense mechanisms into account to maintain precision and relevance. However, given that static rules are typically fixed and labor-intensive to develop, and that defense mechanisms often vary in form and do not follow consistent patterns, we believe that static analysis tools could benefit from integrating complementary techniques—such as the powerful semantic understanding capabilities of large language models (LLMs)—to reduce false positives while preserving soundness.
- **Explore multi-tool integration to improve coverage.** RQ3 reveals significant inconsistency across tools, suggesting that they detect largely disjoint sets of vulnerabilities. Developers can leverage our evaluation to select optimal tool combinations for specific versions and vulnerability types—for example, Vandal for REN. This highlights the potential value of integrating complementary detection strategies—possibly using emerging

techniques such as large language models—into a unified tool to achieve broader and more reliable vulnerability coverage. Additionally, based on the detection results, exploring the migration of capabilities from tools with poor compatibility to newer versions using learning-based methods is a promising direction. This approach can help researchers preserve and extend the core expertise embedded in existing tools.

7.3 Threats to Validity

Internal. (1) Due to the absence of ground truth labels in real-world contract datasets, the manual confirmation is not immune to human error. We established well-defined audit rules to mitigate this risk. Furthermore, the labels are publicly accessible to enable the verification of the results by other researchers. (2) While our toolset is derived from existing surveys and may exclude the latest tools, this is aligned with our focus on version evolution, where earlier tools are more representative. (3) The set of selected vulnerabilities does not cover all those detectable by the tools. However, during our experiment, we obtained the detection results for all vulnerabilities supported by the tools and made them publicly available, enabling further analysis for researchers interested in other vulnerability types. (4) Our consistency analysis is based on raw detection results without ground-truth labels, which inevitably include many false positives. While this limits precision, it still reveals a key insight: the tools tend to be complementary rather than conflicting, regardless of whether the results are true or false positives. (5) Unlike traditional software, smart contracts are immutable once deployed, meaning that multiple upgradable versions of the same contract are not available. Additionally, automated upgrades fall outside the scope of this study, making evaluations based on different versions of the same contract infeasible.

External. (1) Some evaluated tools, such as Mythril and Slither, undergo continuous updates, which may cause our findings to differ from those produced by the latest versions. To minimize this risk, we provide the exact tool versions used in our study to ensure reproducibility, and we have made the execution scripts publicly accessible to help researchers easily obtain results from the latest tool versions. (2) Smart contract development often involves code reuse, particularly through open-source libraries. While this results in shared code across contracts, widely used libraries follow secure development practices and act as standardized interfaces, not major sources of vulnerabilities. Since security risks mainly arise from contract-specific implementations, the impact of code reuse on our analysis is minimal.

8 Conclusion

In this paper, we present an empirical evaluation of 22 static analysis tools on a representative sample of real-world Ethereum contracts, and further assess 10 tools—compatible across Solidity versions 0.4.x to 0.8.x—on a large dataset of 251,340 contracts. Our study reveals that more than half (12) of the evaluated tools are not fully compatible with all Solidity versions, primarily due to their inability to accommodate syntax changes and increasing code complexity in newer versions. Among the ten tools compatible with all versions, nine fail to maintain stable detection rates, five exhibit substantial fluctuations in true positive rates, and seven demonstrate clear effectiveness degradation, indicating limited adaptability despite successful execution. Additionally, tool consistency significantly decreases with newer versions, highlighting growing divergence in detection behavior. These findings emphasize the complementary nature of current tools and the potential benefit of combining them. Finally, we analyze the root causes of tool limitations and identify semantic drift in contract development over time, offering valuable insights for improving future research.

Acknowledgment

This work was supported in part by National Natural Science Foundation of China under Grant 61972373, the Basic Research Program of Jiangsu Province under Grant BK20201192, the Anhui Provincial Department of Science and Technology under Grant 202103a05020009. The research of Dr. Xue is also supported by CAS Pioneer Hundred Talents Program of China.

References

[1] 2024. Open source data for this paper. https://github.com/gcf3711/EmpiricalSCST Commit ID: 34de0244abfcc9dca97752aa411b512c94d4ab10.
[2] 2024. Semgrep. https://github.com/Decurity/semgrep-smart-contracts?tab=readme-ov-file Online; accessed 29 June 2024.
[3] 2024. Smashing Smart Contracts. https://github.com/muellerberndt/smashing-smart-contracts/blob/master/smashing-smart-contracts-1of1.pdf Online; accessed 29 June 2024.
[4] 2024. Solhint. https://github.com/protofire/solhint/blob/develop/docs/rules/security/reentrancy.md Online; accessed 29 June 2024.
[5] Al Danial. 2024. Count Lines of Code. https://github.com/AlDanial/cloc Online; accessed 29 June 2024.
[6] antlr. 2024. ANTLR v4. https://github.com/antlr/antlr4 Online; accessed 17 July 2024.
[7] ApeWorX Ltd. 2024. py-solc-x. https://pypi.org/project/py-solc-x/#history Online; accessed 29 June 2024.
[8] BlockSec. 2022. When "SafeMint" Becomes Unsafe: Lessons from the HypeBears Security Incident. https://blocksecteam.medium.com/when-safemint-becomes-unsafe-lessons-from-the-hypebears-security-incident-2965209bda2a Online; accessed 29 June 2024.
[9] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 454–469. doi:10.1145/3385412.3385990
[10] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981* (2018).
[11] Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander. 2013. Density-based clustering based on hierarchical density estimates. In *Pacific-Asia conference on knowledge discovery and data mining*. Springer, 160–172.
[12] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Benjamin Livshits. 2024. Smart Contract and DeFi Security Tools: Do They Meet the Needs of Practitioners?. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.
[13] Jiachi Chen, Yiming Shen, Jiashuo Zhang, Zihao Li, John C. Grundy, Zhenzhe Shao, Yanlin Wang, Jiashui Wang, Ting Chen, and Zibin Zheng. 2025. FORGE: An LLM-driven Framework for Large-Scale Smart Contract Vulnerability Dataset Construction. *ArXiv* abs/2506.18795 (2025). https://api.semanticscholar.org/CorpusID:279999749
[14] Yuchiro Chinen, Naoto Yanai, Jason Paul Cruz, and Shingo Okamura. 2020. RA: Hunting for Re-Entrancy Attacks in Ethereum Smart Contracts via Static Analysis. In *2020 IEEE International Conference on Blockchain (Blockchain)*. 327–336. doi:10.1109/Blockchain50366.2020.00048
[15] Consensys. 2024. A complete suite of trusted products to build anything in web3. https://consensys.io/ Online; accessed 29 June 2024.
[16] DHS and CISA. 2022. CVE Website. https://cve.mitre.org/. Online; accessed 1 January 2022.
[17] Monika di Angelo, Thomas Durieux, João Fernando Ferreira, and Gernot Salzer. 2023. Evolution of automated weakness detection in Ethereum bytecode: a comprehensive study. *Empirical Software Engineering* 29 (2023). https://api.semanticscholar.org/CorpusID:257631744
[18] M. di Angelo, T. Durieux, J. F. Ferreira, and G. Salzer. 2023. SmartBugs 2.0: An Execution Framework for Weakness Detection in Ethereum Smart Contracts. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 2102–2105. doi:10.1109/ASE56229.2023.00060
[19] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*. 530–541.
[20] Ethereum. 2024. Ethereum Improvement Proposals. https://eips.ethereum.org/ Online; accessed 17 July 2024.
[21] Ethereum. 2024. The history of Ethereum. https://ethereum.org/en/history/ Online; accessed 17 July 2024.
[22] Etherscan. 2024. Ethereum Daily Verified Contracts Chart. https://etherscan.io/chart/verified-contracts Online; accessed 29 June 2024.
[23] Josselin Feist, Gustavo Greico, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain* (Montreal, Quebec, Canada) *(WETSEB '19)*. IEEE Press, 8–15. doi:10.1109/WETSEB.2019.00008
[24] Asem Ghaleb and Karthik Pattabiraman. 2020. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 415–427.
[25] 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*.
[26] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2023. AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*.

[27] Puneet Gill, Indrani Ray, Alireza Lotfi Takami, and Mahesh Tripunitara. 2022. Finding unchecked low-level calls with zero false positives and negatives in Ethereum smart contracts. In *International Symposium on Foundations and Practice of Security*. Springer, 305–321.

[28] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: surviving out-of-gas conditions in Ethereum smart contracts. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 116 (oct 2018), 27 pages. doi:10.1145/3276486

[29] Tianyuan Hu, Jingyue Li, André Storhaug, and Bixin Li. 2023. Why Smart Contracts Reported as Vulnerable were not Exploited? *Authorea Preprints* (2023).

[30] QingYuan Huang, ZiXun Zeng, and Ying Shang. 2024. An Empirical Study of Integer Overflow Detection and False Positive Analysis in Smart Contracts. In *Proceedings of the 2024 8th International Conference on Big Data and Internet of Things*. 247–251.

[31] Sungjae Hwang and Sukyoung Ryu. 2020. Gap between theory and practice: an empirical study of security patches in solidity. 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, 542–553. doi:10.1145/3377811.3380424

[32] Paul Jaccard. 1901. Etude de la distribution florale dans une portion des Alpes et du Jura. *Bulletin de la Societe Vaudoise des Sciences Naturelles* 37 (01 1901), 547–579. doi:10.5169/seals-266450

[33] Chihiro Kado, Naoto Yanai, Jason Paul Cruz, and Shingo Okamura. 2023. An Empirical Study of Impact of Solidity Compiler Updates on Vulnerabilities. In *2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. 92–97. doi:10.1109/PerComWorkshops56833.2023.10150389

[34] Ken Domino. 2024. Grammars-v4. https://github.com/antlr/grammars-v4 Online; accessed 17 July 2024.

[35] 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/

[36] Kaixuan Li, Yue Xue, Sen Chen, Han Liu, Kairan Sun, Ming Hu, Haijun Wang, Yang Liu, and Yixiang Chen. 2024. Static Application Security Testing (SAST) Tools for Smart Contracts: How Far Are We? *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1447–1470.

[37] Ye Liu, Yi Li, Shang-Wei Lin, and Cyrille Artho. 2022. Finding permission bugs in smart contracts with role mining. *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (2022). https://api.semanticscholar.org/CorpusID:250163928

[38] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. doi:10.1145/2976749.2978309

[39] Leland McInnes, John Healy, and James Melville. 2018. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426* (2018).

[40] NCC Group. 2024. Decentralized Application Security Project. https://dasp.co/ Online; accessed 29 June 2024.

[41] OpenAI. 2025. text-embedding-3-large. https://platform.openai.com/docs/models/text-embedding-3-large Online; accessed 1 June 2025.

[42] OpenZeppelin. 2018. SafeMath.sol. https://github.com/ConsenSysMesh/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol Online; accessed 29 June 2024.

[43] Martin Ortner and Shayan Eskandari. [n. d.]. Smart Contract Sanctuary. ([n. d.]). https://github.com/tintinweb/smart-contract-sanctuary

[44] Ortner, Martin and Eskandari, Shayan. 2024. Smart Contract Sanctuary. https://github.com/tintinweb/smart-contract-sanctuary-ethereum/tree/575175ca679698ddbaef3c2c937df4cecadfc97e Online; accessed 29 June 2024.

[45] Daniel Perez and Benjamin Livshits. 2021. Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1325–1341. https://www.usenix.org/conference/usenixsecurity21/presentation/perez

[46] 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? *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 566–579. doi:10.1145/3460319.3464837

[47] Schloss Dagstuhl. 2024. The dblp computer science bibliography. https://dblp.org/ Online; accessed 29 June 2024.

[48] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. ethor: Practical and provably sound static analysis of ethereum smart contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 621–640.

[49] SmartContractSecurity. 2024. Smart Contract Weakness Classification. https://swcregistry.io Online; accessed 29 June 2024.

[50] 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.

[51] Solidity Authors. 2024. source mappings. https://docs.soliditylang.org/en/v0.6.0/miscellaneous.html?highlight=source%20mapping#source-mappings Online; accessed 29 June 2024.

[52] Zlatko Stapic, Eva García López, Antonio García Cabot, Luis de Marcos Ortega, and Vjeran Strahonja. 2012. Performing systematic literature review in software engineering. In *Central European Conference on Information and Intelligent Systems*. Faculty of Organization and Informatics Varazdin, 441.

[53] The Solidity Authors. 2023. Security Considerations. https://docs.soliditylang.org/en/latest/security-considerations.html#use-the-checks-effects-interactions-pattern Online; accessed 29 June 2024.

[54] The Solidity Authors. 2024. Checked or Unchecked Arithmetic. https://docs.soliditylang.org/en/latest/control-structures.html#checked-or-unchecked-arithmetic Online; accessed 29 June 2024.

[55] The Solidity Authors. 2024. Solidity Documentation. https://docs.soliditylang.org/en/latest/ Online; accessed 17 July 2024.

[56] Zhenzhou Tian, Jie Tian, Zhongmin Wang, Yanping Chen, Hong Xia, and Lingwei Chen. 2022. Landscape estimation of solidity version usage on ethereum via version identification. *International Journal of Intelligent Systems* 37, 1 (2022), 450–477.

[57] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 9–16.

[58] tintinweb. 2024. solidity-parser. https://pypi.org/project/solidity-parser/#history Online; accessed 29 June 2024.

[59] Tom Preston-Werner, inventor of Gravatar and cofounder of GitHub. 2024. Semantic Versioning 2.0.0. https://semver.org/ Online; accessed 29 June 2024.

[60] 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* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 664–676. doi:10.1145/3274694.3274737

[61] Christof Ferreira Torres, Mathis Steichen, and Radu State. 2019. The art of the scam: demystifying honeypots in ethereum smart contracts. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 1591–1607.

[62] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018). https://api.semanticscholar.org/CorpusID:46936025

[63] Li Zhang, Jia-Hao Tian, Jing Jiang, Yi-Jun Liu, Meng-Yuan Pu, and Tao Yue. 2018. Empirical research in software engineering—a literature survey. *Journal of Computer Science and Technology* 33 (2018), 876–899.

[64] Pengcheng Zhang, Feng Xiao, and Xiapu Luo. 2019. Soliditycheck: Quickly detecting smart contract problems through regular expressions. *arXiv preprint arXiv:1911.09425* (2019).

[65] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2020. Smart Contract Vulnerability Detection using Graph Neural Network. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, Christian Bessiere (Ed.). International Joint Conferences on Artificial Intelligence Organization, 3283–3290. doi:10.24963/ijcai.2020/454 Main track.