SCAR: Mining and Structuring Smart Contract Security Audit Reports

Ilham Qasse
ilham20@ru.is
Reykjavik University
Reykjavik, Iceland

Po-Yu Tseng
tsengben01@gmail.com
National Taiwan University
Taipei, Taiwan

Mohammad Hamdaqa
mhamdaqa@polymtl.ca
Polytechnique Montreal
Montreal, Canada

Gísli Hjálmtýsson
gisli@ru.is
Reykjavik University
Reykjavik, Iceland

Abstract

Smart contract security audit reports contain rich information about vulnerabilities and code quality issues in Web3 projects. However, these reports are scattered across different sources and formats, making large-scale analysis difficult. We present SCAR (Smart Contract Audit Repository), an open-source dataset and tool that automatically aggregates these audit reports. SCAR crawls reports from leading security firms (e.g., OpenZeppelin) and community contests (e.g., Code4rena), parses them into a structured JSON schema, and offers a queryable API for accessing the data. Its pipeline includes a crawler, a text-mining module to standardize findings (e.g., vulnerability types, severity, code references), and a web API for retrieving insights. With hundreds of audits covering thousands of issues, SCAR enables empirical studies of smart contract vulnerabilities at scale. The SCAR project repository is available on GitHub, and the screencast demo is available at this link.

CCS Concepts

• **Security and privacy** → **Software security engineering**; *Software and application security*; • **Software and its engineering** → *Software libraries and repositories.*

Keywords

smart contracts, security audits, audit reports, dataset, decentralized finance, empirical study

ACM Reference Format:
Ilham Qasse, Po-Yu Tseng, Mohammad Hamdaqa, and Gísli Hjálmtýsson. 2026. SCAR: Mining and Structuring Smart Contract Security Audit Reports. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 05–09, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3803437.3806435

FSE Companion '26, Montreal, QC, Canada

ACM ISBN 979-8-4007-2636-1/2026/07
https://doi.org/10.1145/3803437.3806435

1 Introduction

Smart contracts are self-executing programs deployed on blockchains such as Ethereum to enforce agreements without intermediaries [18]. Their transparent and immutable nature has led to widespread use across decentralized finance (DeFi), token issuance and decentralized autonomous organisations. However, immutability also means that software bugs and security flaws are difficult to patch once code is deployed. Exploits have led to substantial losses, including the infamous DAO hack in 2016 when attackers drained approximately 3.6 million Ether (about $50 million at the time) from a smart contract [15]. More recent incidents involve cross-chain bridge exploits and flash-loan attacks that result in multimillion-dollar losses [14]. Over the past decade many tools have been developed to detect vulnerabilities in smart contracts. These include static analyzers such as *Slither* [4], symbolic execution engines like Mythril [2] and Manticore [19], formal verifiers (for example, KEVM [8] and VerX [13]) and property-based fuzzers such as Echidna [6]. Despite this progress, recent empirical studies report low precision and recall, inconsistent coverage and poor alignment between tool warnings and real-world vulnerabilities [3, 7, 10, 12, 20]. Because of these shortcomings, developers often rely on manual security audits conducted by expert firms. Auditors identify subtle logic flaws and design issues that automated tools miss, and their reports include detailed explanations of findings, severity levels, possible exploit scenarios and remediation advice.

Nevertheless, audit reports are scattered across different websites and stored in incompatible formats. Providers publish them as PDF files, web pages, Markdown documents or GitHub repositories. Terminology and structure differ significantly across firms, which makes it difficult to compare findings, identify recurring issues or build benchmarks for research. Existing datasets tend to concentrate on tool-generated alerts or synthetic bugs rather than real audit findings [5, 9, 16, 21, 23]. Even datasets that contain findings from audit reports often normalize the data into broad categories and are static snapshots that are not updated as new audits are published. To address these challenges, this paper introduces **SCAR**, the *Smart Contract Audit Repository*. SCAR collects audit reports from multiple professional platforms and community contests, parses them into a unified schema and makes the data available for research. Unlike previous efforts, SCAR continuously crawls fresh reports, preserves the full text of each finding (including the original wording, severity and remediation status) and provides an extensible

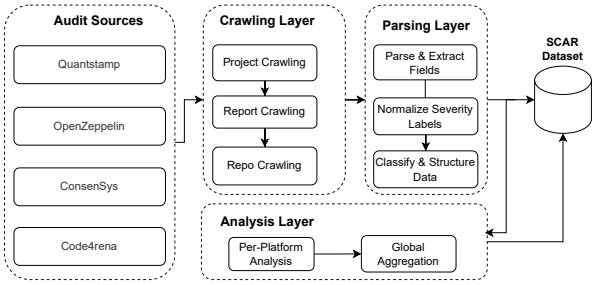

Figure 1: SCAR architecture

pipeline for normalizing and analyzing the data. Our contributions are summarized as follows:

- We develop a modular toolchain that crawls audit reports from four leading providers (Quantstamp, ConsenSys Diligence, OpenZeppelin and Code4rena), extracts structured data from heterogeneous formats and stores it in a uniform JSON schema.
- We define a schema that captures project metadata, audit scope descriptions, vulnerability findings, categories, severity and remediation status. Platform-specific information (for example, update histories or community vote counts) is retained in extension fields.
- We release a growing dataset of more than a thousand audited projects and nearly fifteen thousand individual findings. Statistical summaries by platform, severity and programming language are provided.

By bridging the gap between informal, manually written audit documents and structured machine-readable datasets, SCAR enables more reproducible research, data-driven tool development, and transparency in Web3 security practices.

2 SCAR Architecture

SCAR is designed as a modular pipeline that automates the end-to-end processing of smart contract audit reports. Its architecture consists of three main layers: Crawling, Parsing, and Analysis. Each layer contains components tailored to the format and content conventions of the four supported auditing platforms: **Quantstamp**[1], **ConsenSys Diligence**[2], **OpenZeppelin**[3], and **Code4rena**[4]. These platforms were selected for their public availability, large number of reports, and stylistic diversity. For instance, Code4rena reports aggregate community-submitted issues while Quantstamp follows a fixed structure with summary tables and update histories. SCAR handles these differences using a layered architecture, illustrated in Figure 1.

Crawling Layer. SCAR implements three types of crawlers per platform. Project crawlers collect basic metadata such as project names, tags, audit dates, and report links. Report crawlers retrieve the actual audit content, which may be structured JSON, web pages, Markdown documents, or static PDFs depending on the platform. Repository crawlers extract GitHub references and fetch source code files or links from each report when available. All crawlers inherit from shared base classes and can be launched in parallel to improve throughput. Selenium is used to handle dynamically rendered content, especially in platforms like Quantstamp that serve client-side JSON through the browser.

Parsing Layer. Each audit platform has its own parser, responsible for transforming heterogeneous report formats into the SCAR unified schema. The parsers extract key elements such as issue titles, severity levels, descriptions, recommendations, and remediation status. They also handle semi-structured content, including code snippets, nested subsections, blockquotes, and external links. These are stored as structured fields in the resulting JSON files. Since platforms use inconsistent terminology, SCAR normalizes these categories through a central configuration file, ensuring that downstream statistics remain comparable across sources.

Analysis Layer. The final layer analyzes the normalized data to produce structured summaries. Each platform has its own analyzer class, which computes per-project statistics including issue counts by severity, remediation status, programming languages mentioned (inferred from file extensions), and counts of external links. These outputs are written to platform-specific JSON files and then merged by a statistics module that aggregates global totals. This final output supports cross-platform comparisons and can be used to generate visualizations or benchmarks.

2.1 Implementation and Availability

SCAR is implemented in Python. Each platform-specific module inherits from common base classes and is dynamically selected using a factory pattern. The entire pipeline can be executed via a command-line interface or through Docker. The crawler modules use headless Chrome for web automation and download management. The parsing and analysis modules follow the same directory and file layout across all sources, which simplifies future extensions to additional audit providers. The tool and dataset are available at: https://github.com/IlhamQasse/SCAR.git

3 Dataset overview

The SCAR dataset is an integrated repository of smart contract audit reports. This section outlines the schema and provides summary statistics.

3.1 Dataset schema

SCAR stores data using a relational schema, shown in Figure 2. The `project` table holds unique identifiers and names, while the `report` table links each report to a project along with the platform, source URL, date, and local storage path. The `repo` table lists referenced code repositories. Audit scopes are stored in the `scope` table. Individual findings are represented in `unified_issue`, which records severity, status, descriptive text, exploit scenario, and recommendation. Platform-specific extension tables preserve additional fields. Enumeration types define the audit `platform`, a normalized set of `severity` categories (critical, high, medium, low, and informational) and issue `_status` values (open, acknowledged, and fixed). This schema supports consistent querying across providers.

[1] https://certificate.quantstamp.com
[2] https://consensys.io/diligence/audits
[3] https://blog.openzeppelin.com/tag/security-audits
[4] https://code4rena.com/reports

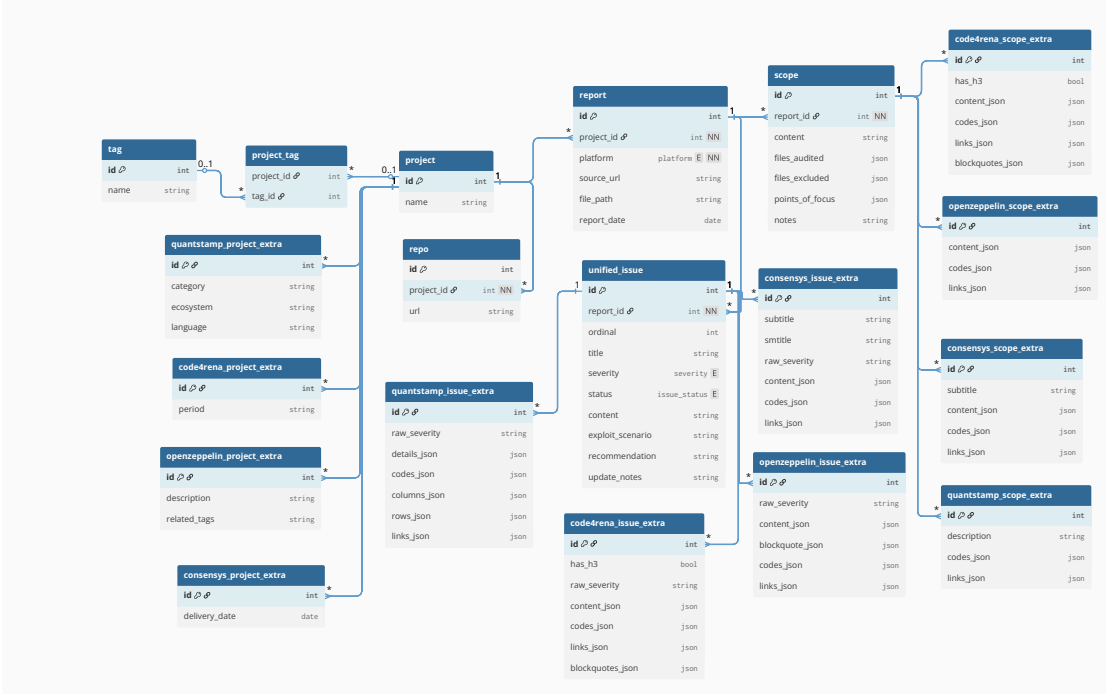

Figure 2: SCAR schema.

Table 1: Distribution of audited projects and vulnerabilities across platforms.

Platform	Projects	Website	GitHub repo	PDF	Vulnerabilities
Quantstamp	360	211	0	149	4 367
ConsenSys	158	137	20	1	1 295
OpenZeppelin	225	225	0	0	2 313
Code4rena	386	386	0	0	6 980

3.2 Dataset composition

The current release aggregates 1 129 audited projects across the four platforms. Table 1 summarizes the number of audited projects per platform, broken down by report medium and accompanied by the total number of vulnerabilities. Code4rena and Quantstamp contribute the largest numbers of reports and findings, whereas OpenZeppelin's 225 reports are all hosted on the provider's website. Across the entire corpus SCAR records 14 955 distinct vulnerabilities.

3.3 Programming-language distribution

We record the programming languages mentioned in each report by counting how often a file with the given extension appears across all reports for each platform. Solidity dominates with tens of thousands of occurrences in Code4rena reports and similar numbers in the other platforms. Rust and Go (Golang) appear frequently in Quantstamp and OpenZeppelin audits, while TypeScript is common in Quantstamp and ConsenSys reports. Table 2 lists the counts per language.

Table 2: Frequency of programming-language audited per platform.

Language	Quantstamp	ConsenSys	OpenZeppelin	Code4rena
Solidity	42 973	7 375	52 809	94 541
Golang	794	215	362	1 991
Rust	2 384	0	1 341	1 958
Vyper	43	22	32	83
Cadence	114	0	0	3
TypeScript	5 409	679	329	289
JavaScript	3 339	233	98	180
Python	84	33	22	49
Circom	200	0	228	0

3.4 Vulnerability severity

Severity distributions vary across platforms because each provider employs its own taxonomy. Quantstamp defines five severity categories: High, Medium, Low, Informational, and Undetermined. ConsenSys uses four levels: Critical, Major, Medium, and Minor. OpenZeppelin reports Critical, High, Medium, and Low issues. Code4rena distinguishes High, Medium, Low, and a separate Non-critical Low risk category. Table 3 shows the number of findings in each category for all four platforms. Quantstamp and Code4rena audits account for the largest share of issues (4 367 and 6 980 respectively, as shown in Table 1). If we normalize provider-specific severities into three broad groups: *High* (Critical or High), *Medium* (Major/Medium/Minor), and *Low* (Low, Informational, Undetermined), then across the entire dataset about 12.4% of the 14 955 findings are high severity,

Table 3: Issue counts by severity category for each platform.

Platform	Severity	Count
Quantstamp	High risk	366
	Medium risk	726
	Low risk	1643
	Informational	1366
	Undetermined	266
ConsenSys	Critical risk	92
	Major risk	243
	Medium risk	415
	Minor risk	545
OpenZeppelin	Critical risk	121
	High risk	215
	Medium risk	591
	Low risk	1386
Code4rena	High risk	1054
	Medium risk	2560
	Low risk	331
	Low risk non critical	3035

34.0% are medium, and 53.7% are low. This analysis highlights that severe vulnerabilities are relatively rare compared to lower-level issues.

4 Related Work

Smart contract vulnerability datasets vary in purpose and quality depending on how they are constructed. Several projects focus on benchmarking detection tools using curated or synthetic examples. SmartBugs [5] provides a suite of known-vulnerable contracts for evaluating static analyzers and symbolic execution engines. SC-Bench [21] injects thousands of synthetic ERC-standard violations into real contracts. While useful for stress-testing tools, injected bugs do not necessarily reflect the subtle logic flaws that appear in real audits. Other datasets focus on specific vulnerability classes. SCRUBD [22] targets reentrancy and exception-handling issues by combining two sources: real vulnerable contracts collected from the ecosystem and manually synthesized corner-case examples. These cases are crafted by analyzing reentrancy scenarios rather than relying on static tool outputs. In contrast, AutoMESC [17] labels metadata-related flaws using static analysis tools. Such tool-labeled datasets inherit the limitations and biases of the analyzers they depend on and therefore do not capture expert audit reasoning or remediation practices.

Because of these limitations, recent work has shifted toward mining vulnerability data directly from professional audit reports. DApp-Scan [23] aggregates over a thousand public audits, identifying and labeling weakness instances based on report excerpts. FORGE [1] uses a large language model to extract vulnerability descriptions from unstructured audit text and map them to CWE taxonomies. Smart Contract VulnDB [11] scrapes audit reports into a structured JSON database of issues. While these approaches help build large-scale vulnerability catalogs, they typically extract only summary fields such as titles and severities. They omit contextual elements like audit scope, project goals, tooling choices, exploitation scenarios, or fix explanations. They also treat reports as static inputs and do not support incremental updates or audit versioning.

SCAR builds upon these ideas but differs along several key dimensions. First, SCAR operates on complete audit reports from multiple professional platforms and preserves the full structure and wording of each finding, including severity labels, affected code, remediation status, and proposed fixes. Second, SCAR supports reports in diverse formats (Markdown, HTML, PDF) and applies source-specific parsing logic to normalize their contents into a unified schema. Third, SCAR is continuously updated through automated crawling and parsing, allowing it to track changes over time and include newly published audits as they appear. These features make SCAR well-suited for downstream applications that require high-fidelity audit narratives, such as ML fine-tuning, longitudinal analysis, or explainable evaluation of security tools.

5 Applications of SCAR

SCAR has several applications for researchers, developers, auditors, projects, and regulators.

Empirical studies. SCAR enables large-sample longitudinal analyzes of smart contract vulnerabilities. Researchers can ask which vulnerability types dominate by ecosystem, language or project category. They can also analyze how often high-severity issues are fully remediated and whether repeated audits reduce outstanding issues. Because SCAR continuously tracks reports it supports trend analysis rather than one-off snapshots.

Tool improvement. Tool builders need realistic labelled examples to train detectors and reduce false positives. SCAR provides thousands of expert-identified findings and remediation narratives. These can be used for supervised learning, retrieval-augmented auditing (surfacing similar past findings during review) and learning-to-repair from {finding, fix} pairs.

Co-auditing. SCAR's history-aware data can support co-auditing workflows that keep human auditors central. For example, similarity search over prior findings can seed checklists. Active learning can refine mapping between vendor severity labels. Dashboards can reveal disagreement patterns across firms.

Education and governance. Authentic remediation narratives grounded in audit reports are valuable for teaching secure development practices. Projects and decentralized organisations can use SCAR to justify risk decisions and reference comparable findings. Regulators and foundations may use SCAR to set evidence-based minimum standards for audits.

6 Conclusion and future work

We presented SCAR, a dataset and toolchain for structuring smart contract audit reports. SCAR continuously crawls reports from multiple providers, parses them into a unified schema and releases a growing dataset of projects and findings. By preserving the full text and contextual information of audit reports, SCAR enables reproducible research, data-driven tool development and transparency in Web3 security practices. Future work will add additional providers, incorporate finer grained language and project features and provide a public search interface.

References

[1] Jiachi Chen, Yiming Shen, Jiashuo Zhang, Zihao Li, John 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 preprint* arXiv:2506.18795 (June 2025). doi:10.48550/arXiv.2506.18795 Accessed 17 Nov 2025.

[2] ConsenSys. 2023. Mythril Security Analysis Tool. GitHub repository. https://github.com/ConsenSys/mythril Accessed 17 Nov 2025.

[3] 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 2020 IEEE/ACM International Conference on Automated Software Engineering*. IEEE/ACM, 530–541. doi:10.1145/3377811.3380364 Accessed 17 Nov 2025.

[4] Josselin Feist, Gustavo Grieco, 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 (WETSEB '19)*. IEEE/ACM, 8–15. doi:10.1109/WETSEB.2019.00010 Accessed 17 Nov 2025.

[5] João F. Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2020. SmartBugs: A Framework to Analyze Solidity Smart Contracts. In *Proceedings of the 2020 IEEE/ACM International Conference on Automated Software Engineering*. IEEE/ACM, 1876–1880. doi:10.48550/arXiv.2007.04771 Accessed 17 Nov 2025.

[6] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*. ACM, 149–160. doi:10.1145/3395363.3404366 Accessed 17 Nov 2025.

[7] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Principles of Security and Trust (POST) 2018*. Lecture Notes in Computer Science, Vol. 10804. Springer, 243–269. doi:10.1007/978-3-319-89722-6_10 Accessed 17 Nov 2025.

[8] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Ştefănescu, and Grigore Roşu. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *Proceedings of the 2018 IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 204–217. doi:10.1109/CSF.2018.00022 Accessed 17 Nov 2025.

[9] Xiaoqi Li, Peng Liao, Junjie Li, Fangming Liu, Hui Sun, and Wei Wang. 2021. A Survey on Blockchain Smart Contracts: Vulnerabilities, Attacks, and Tools. *Future Generation Computer Systems* 119 (2021), 45–77. doi:10.1016/j.future.2021.01.012 Accessed 17 Nov 2025.

[10] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 2018 Annual Computer Security Applications Conference (ACSAC)*. ACM, 653–663. doi:10.1145/3274694.3274743 Accessed 17 Nov 2025.

[11] Martin Ortner. 2025. Smart Contract VulnDB. GitHub repository. https://github.com/tintinweb/smart-contract-vulndb Open dataset of audit issues; JSON snapshots and daily updates. Accessed 17 Nov 2025.

[12] Reza M. Parizi, Ali Dehghantanha, Kim-Kwang Raymond Choo, and Amritraj Singh. 2018. Empirical Vulnerability Analysis of Automated Smart Contracts Security Testing on Blockchains. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering (CASCON)*. IBM Corp., 103–113. doi:10.48550/arXiv.1809.02702 Accessed 17 Nov 2025.

[13] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1661–1677. doi:10.1109/SP40000.2020.00024 Accessed 17 Nov 2025.

[14] Kaihua Qin, Liyi Zhou, Benjamin Livshits, and Arthur Gervais. 2020. Attacking the DeFi Ecosystem with Flash Loans for Fun and Profit. *arXiv preprint* arXiv:2003.03810 (2020). https://arxiv.org/abs/2003.03810 Accessed 17 Nov 2025.

[15] Jason Scharfman. 2024. Decentralized autonomous organization (dao) fraud, hacks, and controversies. In *The Cryptocurrency and Digital Asset Fraud Casebook, Volume II: DeFi, NFTs, DAOs, Meme Coins, and Other Digital Asset Hacks*. Springer, 65–106.

[16] Smart Contract Weakness Classification Registry. 2025. SWC Registry. Web resource. https://swcregistry.io/ Accessed 17 Nov 2025.

[17] Majd Soud, Ilham Qasse, Grischa Liebel, and Mohammad Hamdaqa. 2023. Automesc: Automatic framework for mining and classifying ethereum smart contract vulnerabilities and their fixes. In *2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 410–417.

[18] Nick Szabo. 1997. Formalizing and Securing Relationships on Public Networks. *First Monday* 2, 9 (September 1997), —. doi:10.5210/fm.v2i9.548 Accessed 17 Nov 2025.

[19] Trail of Bits. 2023. Manticore: Symbolic Execution Tool for Analyzing Smart Contracts and Binaries. GitHub repository. https://github.com/trailofbits/manticore Accessed 17 Nov 2025.

[20] Yilin Wang, Xiangping Chen, Yuan Huang, Hao-Nan Zhu, Jing Bian, and Zibin Zheng. 2023. An Empirical Study on Real Bug Fixes from Solidity Smart Contract Projects. *Journal of Systems and Software* 204 (2023), 111787. doi:10.1016/j.jss.2023.111787 Accessed 17 Nov 2025.

[21] Shihao Xia, Mengting He, Linhai Song, and Yiying Zhang. 2024. SC-Bench: A Large-Scale Dataset for Smart Contract Auditing. *arXiv preprint* arXiv:2410.06176 (October 2024). https://arxiv.org/abs/2410.06176 Accessed 17 Nov 2025.

[22] Chavhan Sujeet Yashavant, MitrajSinh Chavda, Saurabh Kumar, Amey Karkare, and Angshuman Karmakar. 2024. SCRUBD: Smart Contracts Reentrancy and Unhandled Exceptions Vulnerability Dataset. *arXiv preprint* arXiv:2412.09935 (December 2024). doi:10.48550/arXiv.2412.09935 Accessed 17 Nov 2025.

[23] Zibin Zheng, Jiachi Chen, Jianzhong Su, Zhijie Zhong, Mingxi Ye, and David Lo. 2024. DAppSCAN: Building Large-Scale Datasets for Smart Contract Weaknesses in DApp Projects. *IEEE Transactions on Software Engineering* 50, 6 (2024), 1360–1373. doi:10.1109/TSE.2023.3240821 Accessed 17 Nov 2025.