Scientific Data

https://doi.org/10.1038/s41597-026-07025-5

Article in Press

DIVE: A Multi-Label Smart Contract Vulnerability Dataset

Received: 6 October 2025

Accepted: 3 March 2026

Cite this article as: Alsunaidi, S.J., Aljamaan, H., Hammoudeh, M. DIVE: A Multi-Label Smart Contract Vulnerability Dataset. *Sci Data* (2026). https://doi.org/10.1038/s41597-026-07025-5

Shikah J. Alsunaidi, Hamoud Aljamaan & Mohammad Hammoudeh

SCIENTIFIC DATA

CONFIDENTIAL
COPY OF SUBMISSION FOR PEER REVIEW ONLY

Tracking no: SDATA-25-05661A

DIVE: A Multi-Label Smart Contract Vulnerability Dataset

Authors: Hamoud Aljamaan (King Fahd University of Petroleum and Minerals), Shikah Alsunaidi (King Fahd University of Petroleum and Minerals), and Mohammad Hammoudeh (King Fahd University of Petroleum and Minerals)

Abstract:
Smart Contract (SC) vulnerabilities are programming errors or design flaws that can lead to financial loss or functional failure, making accurate detection essential. Although Machine Learning (ML) is widely applied to SC vulnerability detection, existing datasets are often small, imbalanced, inconsistently labeled, or nonstandardized, and frequently rely on limited feature representations that do not account for different contract lifecycle stages, restricting generalization and degrading benchmark reliability. This study introduces DIVE, a multi-label dataset that addresses these structural and feature-level limitations. DIVE includes 22,330 real-world SCs deployed between 2016 and 2024, and spanning major Solidity compiler versions, annotated for eight vulnerability types aligned with the Decentralized Application Security Project (DASP) Top 10 taxonomy. It provides 221 pre-deployment and 176 post-deployment features and employs a standardized multi-tool labeling pipeline based on Power-based voting and post-hoc filtering, which corrected 14.3% false positives in DoS and 24.9% in Time Manipulation. Unlike prior datasets, DIVE offers two lifecycle-specific feature sets and an open-source framework enabling reproducible benchmarking and periodic reconstruction aligned with evolving vulnerability patterns.

Datasets:

Repository Name	Dataset Title	Accession Number or DOI	URL to data record	Private reviewer access URL/code
Zenodo	DIVE: A Multi-Label Smart Contract Vulnerability Dataset	10.5281/zenodo.15680735	https://zenodo.org/records/17250330	https://zenodo.org/records/17250330?token=eyJhbGciOiJIUzUxMiJ9.eyJpZCI6Ijg0ZmQ1NDgzLTczZTYtNDNkNC1hNjJlLTYwZGU2NDgyZmJkNSIsImRhdGEiOnt9LCJyYW5kb20iOiIzYWNkZjdkOGMyYjdlMjdlNjczMWNjOGIxYmIyMzU0NyJ9.eZI5XIJ9xlYroQWo01q_BvFGZcn-bk0-Y3ANH7In39pqJckoB-69EjAU_59aYHEwczF4H4gENhXOtVS9wTjlMA

DIVE: A Multi-Label Smart Contract Vulnerability Dataset

Shikah J. Alsunaidi[*1], Hamoud Aljamaan[†1,2], and Mohammad Hammoudeh[‡1,3]

[1]Information and Computer Science Department, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia
[2]Interdisciplinary Research Center for Finance and Digital Economy, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia
[3]Interdisciplinary Research Center for Intelligent Secure Systems, King Fahd University of Petroleum and Minerals, Dhahran, 31261, Saudi Arabia

Abstract

Smart Contract (SC) vulnerabilities are programming errors or design flaws that can lead to financial loss or functional failure, making accurate detection essential. Although Machine Learning (ML) is widely applied to SC vulnerability detection, existing datasets are often small, imbalanced, inconsistently labeled, or nonstandardized, and frequently rely on limited feature representations that do not account for different contract lifecycle stages, restricting generalization and degrading benchmark reliability. This study introduces DIVE, a multi-label dataset that addresses these structural and feature-level limitations. DIVE includes 22,330 real-world SCs deployed between 2016 and 2024, and spanning major Solidity compiler versions, annotated for eight vulnerability types aligned with the Decentralized Application Security Project (DASP) Top 10 taxonomy. It provides 221 pre-deployment and 176 post-deployment features and employs a standardized multi-tool labeling pipeline based on Power-based voting and post-hoc filtering, which corrected 14.3% false positives in DoS and 24.9% in Time Manipulation. Unlike prior datasets, DIVE offers two lifecycle-specific feature sets and an open-source framework enabling reproducible benchmarking and periodic reconstruction aligned with evolving vulnerability patterns.

Background & Summary

The emergence of blockchain technology transformed multiple industries by reshaping the ways in which data is created, managed, and shared. At the center of this transformation are SCs, which are self-executing agreements with terms embedded in code[1]. Several ecosystems support SCs, and among them Ethereum is regarded as the benchmark platform because it was the first to offer fully programmable SCs through Solidity[2].

SCs automate processes and reduce reliance on trusted intermediaries, enabling decentralized interactions across both public and private applications[2]. However, security remains a critical concern due to blockchain immutability. Once deployed, SCs cannot be modified without preplanned upgrade mechanisms, leaving vulnerabilities persistent and exploitable[3].

SCs continue to be compromised by multiple categories of vulnerabilities. The DASP taxonomy[4] organizes these into ten groups, which include Reentrancy, Access control flaws, Arithmetic issues, Unchecked low-level calls, Denial of Service (DoS), Bad randomness, Front running, Time manipulation, Short address attacks, and a final category of Unknown issues that contains emerging or not yet classified vulnerabilities.

Various detection approaches emerged in response to these risks. Analysis tools[5] and ML techniques[6] are widely studied, and both can be applied at different stages of the SC lifecycle[2]. As illustrated in Fig. 1, a contract is represented in distinct forms during each stage.

In the creation stage, the contract appears as source code in a high-level programming language, where terms and conditions are formally specified. During deployment, the source code is compiled to generate

[*]shikah.sunaidi@gmail.com
[†]Corresponding author: hjamaan@kfupm.edu.sa
[‡]m.hammoudeh@kfupm.edu.sa

the Application Binary Interface (ABI) and the creation bytecode. These outputs are packaged into a deployment transaction and broadcast to blockchain nodes for validation. Once validated and recorded in a block, the runtime bytecode is assigned a unique contract address and stored in the blockchain state. During execution, the deployed contract responds to external function calls or transactions, which produce state changes or event logs that are permanently recorded in the blockchain ledger. In the final stage, the contract may be explicitly terminated, for example through the selfdestruct operation, or logically deprecated. When terminated, the contract code and state are removed from the active blockchain, although historical records remain preserved.

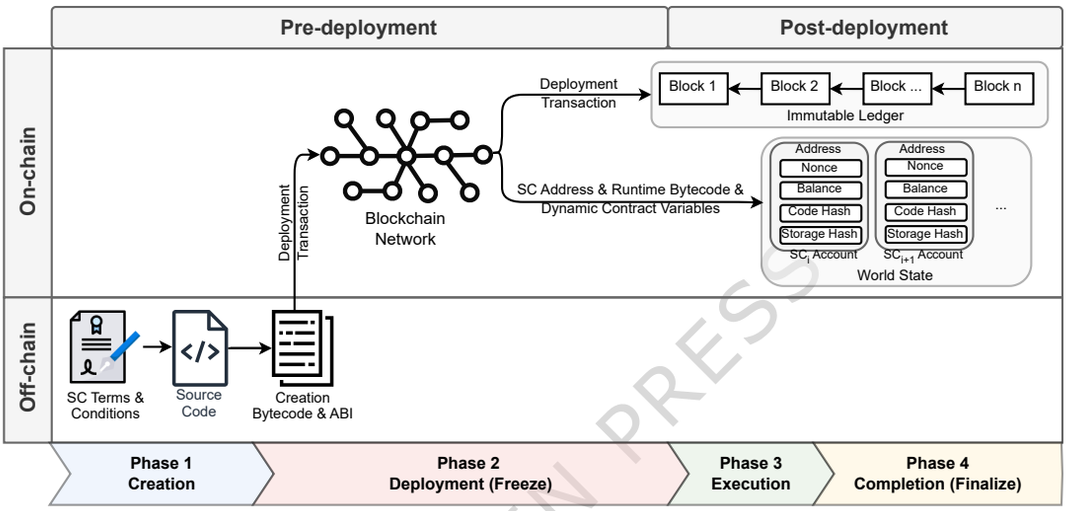


Fig. 1: SC representations across lifecycle stages.

SCs transition from off-chain representations such as source code, ABI, and creation bytecode to on-chain forms. The creation bytecode remains preserved in the deployment transaction history, whereas the runtime bytecode is stored on-chain as static and immutable code. In contrast, contract state variables evolve dynamically during execution, illustrating the distinction between static and dynamic data across the contract lifecycle.

The effectiveness of detection methods depends on access to comprehensive datasets of vulnerable SCs. Although Ethereum data is publicly available [7], the distributed blockchain architecture and the large volume of replicated raw data create significant challenges for data retrieval and exploration. Blockchain explorers such as Etherscan (`https://etherscan.io/`) offer interfaces that support on-chain activity exploration, but their data cannot be used directly to train ML models. Processed and labeled datasets must therefore be developed to enable effective training.

Several datasets were proposed to support research on SC security [8–24]. Although these datasets enabled vulnerability analysis, detection, and benchmarking, they exhibited several limitations. Some relied on single-label annotations [22,24], which did not capture the coexistence of multiple vulnerabilities within a single contract, while others offered limited coverage of real-world contracts [15,17,19,24] or vulnerability categories [9,18,22,24]. In addition, datasets suffered from inconsistencies in vulnerability definitions and label naming, which hindered meaningful comparison across studies and limited the effective reuse of existing datasets [8–12,14–16,20–23]. Furthermore, there was a lack of a comprehensive dataset with rich attributes representing SCs across different lifecycle stages, which limited effective understanding of contract behavior and its evolution. These limitations constrained comprehensive evaluation and hindered the development of robust and generalizable security models.

To address these limitations, this article presents DIVE [25], a multi-label, real-world Ethereum SC dataset designed for security research. DIVE integrates multiple contract representation attributes across different lifecycle stages, including source code, bytecode, opcodes, and transaction-based features, with a preprocessed tabular representation. Contracts are annotated at the source code level using eight vulnerability categories aligned with the DASP Top 10 taxonomy, namely Reentrancy, Access control, Arithmetic, Unchecked low-level calls, DoS, Bad randomness, Front running, and Time manipulation. To further enrich the SC security research domain and support regular dataset updates, this article also

establishes an open-source framework for automated dataset construction, which takes SC addresses as input and produces processed datasets according to configurable settings.

The DIVE dataset supports a wide range of research tasks, including the following.

- Binary, multi-class, multi-label, and multi-task classification. Compared to single-task learning, multi-task classification involves jointly learning multiple related prediction objectives using shared contract representations. For example, a model may predict whether a contract is vulnerable while simultaneously identifying the corresponding vulnerability categories.
- Transfer and domain shift studies across compiler versions, time, and cross-vulnerability settings.
- Representation learning and multi-modal modeling over source code, bytecode/opcodes, and graph structures, e.g., AST/CFG.
- Pre-deployment and post-deployment feature investigation for improving SC vulnerability detection.

Related Work

The literature presents a range of public datasets on SC vulnerabilities. Some cover multiple vulnerability types[8–20], whereas others focus on specific categories such as Reentrancy[24] and phishing[22]. Tables 1–3 summarize prior SC vulnerability datasets across several dimensions, including dataset size, data source, labeling strategies, and feature diversity.

For systematic comparison, these tables are organized around four recurring dimensions in the literature: (i) temporal coverage in terms of deployment years and block numbers, (ii) feature diversity across pre-deployment and post-deployment representations, (iii) the vulnerability taxonomy employed, such as DASP, SWC, or ad hoc labels, and (iv) the labeling methodology, including manual annotation, tool-based analysis, voting schemes, or audit-derived labels.

These dimensions provide a common lens for understanding trade-offs among existing datasets and contextualizing the design choices adopted in DIVE. The remainder of this section examines each dimension in detail.

Dataset Scope and Scale

Dataset sizes vary widely across existing works. Smaller curated datasets typically offer richer annotations compared to larger datasets. Most datasets rely on Etherscan as the primary data source, although some integrate additional repositories such as GitHub or audit reports. Feature availability and the number of supported vulnerability labels vary considerably across datasets, with a limited subset providing automated data collection or labeling pipelines. Collection ranges and deployment periods are often unreported, limiting cross-dataset comparability and reuse.

Contract Attributes in Existing Datasets

The reviewed datasets include several attributes, as shown in Table 2. While most datasets provide a contract address and source code, processed features such as code metrics and transaction-level attributes are often missing. Evaluating and comparing the impact of different feature types on the SC vulnerability detection performance is important but challenging, since no dataset offers a comprehensive feature set. Furthermore, extending an existing dataset with missing features is impractical if essential identifiers, such as the contract address or deployment transaction hash, are unavailable for on-chain extraction.

Vulnerability Labeling Strategies

Labeling approaches can be categorized into three groups: (1) manual labeling, (2) tool-based analysis, and (3) vulnerability injection. Manually labeled datasets are typically limited in size[15,17,19,24], although some mitigate this constraint by employing teams of experienced annotators[14]. In other cases, labels are derived from previously annotated data[15] or audit reports[19]. A few works further combine automated filtering with manual verification, first identifying suspicious samples through predefined patterns and then confirming them manually[10,16].

Tool-based labeling is widely used. Some datasets rely on a single analysis tool, such as MAIAN[9], Slither[12,21], or Oyente[18]. Since no individual tool can detect all vulnerability types, multiple tools are

Table 1: Summary of publicly available datasets on SC Vulnerabilities.

Ref.	Dataset	Data Size				Data Collection			
		Samples	Positives	Att.	L	Source	Range	Autmated Method Available	Utilized In
Tann et al., 2018 [9]	Safe-SCs	920,179	–	2	1	Google Big-Query	From block 0 to 4,799,998	✗	26
Liao et al., 2019 [8]	SoliAudit	17,979	17,609	1	15	Etherscan	–	✗	8,27,28
Ajienka, 2020 [23]	Ajienka's Data	10,476	–	24	1	Etherscan	–	✗	–
Durieux et al., 2020 [17]	SB Curated	69	69	3	10	Etherscan, GitHub, blogs	–	✗	–
Eshghie et al., 2021 [24]	Dynamit	105 T of 25 SC	49T	5	1	Etherscan	–	✓	24
Rossin, 2022 [12]	SlitherAuditedContracts	113k	109,692	3	9	Etherscan, [29]	–	✗	30,31
Yashavan et al., 2022 [13]	ScrawlD	6,780	4,092	1	0	Etherscan	–	✓	32,33
Zhang et al., 2022 [11]	SPCBIG-EC	47,398	35,151	1	1	Ethereum	–	✗	–
Forta 2023, [22]	Token-Impersonation	85,716	375	7	1	Etherscan	–	✗	–
Liu et al., 2023 [14]	SC-Dataset	12,515	–	0	8	Etherscan	–	✗	–
Qian et al., 2023 [10]	SC-Dataset	42,910	–	2	4	Etherscan, GitHub, blogs	–	✗	10,34
Storhaug 2023 [15]	Vulnerable-VerifiedSCs	609	609	12	0	[35]	–	✗	–
Ibba et al., 2024 [21]	SmarthER	47,932	–	23	1	Etherscan, [29]	–	✗	–
Luo et al., 2024 [16]	SCVHunter	47,587	–	2	4	[17]	–	✗	–
Malik, 2024 [20]	Eth-Reputable-Illicit-SC	3,276	191	4	10	–	–	✗	–
Wang et al., 2024 [18]	Contract-GNN	3,001	1,627	1	3	Etherscan	Deployed before January 2024	✗	–
Zheng et al., 2024 [19]	DAppSCAN	682	608	2	1	Etherscan	–	✗	–

Note: Transaction (**T**), Attributes (**Att.**), Labels (**L**).

Table 2: Attributes of publicly available SC vulnerability datasets.

Ref.	Identifiers	Pre-deployment (Code-based)							Post-deployment (Transaction-based)					
	Address	Hash	MD	SC	CM	BC	Op	OpM	DA	C&O	MD	Input	G&F	S
Safe-SCs [9]	•						•							
SoliAudit [8]	•			•			•							
Ajienka's Data [23]	•				•									
SB Curated [17]	•			•						•				
Dynamit [24]		•		•								•	•	•
SlitherAudited Contracts [12]	•			•		•								
ScrawlD [13]	•													
SPCBIG-EC [11]				•										
Token-Impersonation [22]	•		•			•	•							
SC-Dataset [14]				•										
SC-Dataset [10]				•		•								
Vulnerable-VerifiedSCs [15]	•		•	•					•	•				
SmarthER [21]	•			•	•									
SCVHunter [16]	•			•										
Eth-Reputable-Illicit-SC [20]	•			•		•								
Contract-GNN [18]	•													
DAppSCAN [19]				•		•								

Note: Metadata (**MD**) Source code (**SC**), Code metrics (**CM**), Bytecode (**BC**) Opcodes (**Op**), Opcode metrics (**OpM**), Deployment Args (**DA**), Compilation and Optimization (**C&O**), Gas and Fees (**G&F**), Status (**S**).

often combined. For example, SoliAudit [8] utilized Oyente and Remix, SPCBIG-EC [11] applied majority voting across three tools, and ScrawlD [13] relied on majority voting across five tools. However, executing multiple tools is resource-intensive. Moreover, it remains necessary to validate tool accuracy and to design robust consensus mechanisms for confirming vulnerabilities [4].

In addition to labeling existing contracts, vulnerability injection methods insert synthetic vulnerabilities into SC code. For example, SolidiFI [36] analyzes contracts' Abstract Syntax Trees (ASTs) to systematically insert predefined bug patterns.

As shown in Table 3, only a subset of multi-label datasets applies standardized taxonomies. The DASP Top 10 [4] was adopted by several datasets [17,18,24], while SC Weakness Classification (SWC) codes (`https://swcregistry.io/`) was used in others [13,19]. The absence of standardized labels in remaining datasets introduces inconsistencies, as different labels may describe the same vulnerability [37].

Table 3: Overview of dataset labeling strategies in previous works.

Ref.	Labeling Method	Standard Taxonomy	No. of Vulnerabilities
Safe-SCs [9]	MAIAN	–	3
SoliAudit [8]	Oyente & Remix	–	13
Ajienka's Data [23]	–	–	33
SB Curated [17]	Manually	DASP	10
Dynamit [24]	Manually	DASP	1
SlitherAudited Contracts [12]	Slither	–	8
ScrawlD [13]	Majority vote (Mythril, Osiris, Oyente, Slither, SmartCheck)	SWC	8
SPCBIG-EC [11]	Majority vote (Mythril, Oyente, Securify)	–	6
Token-Impersonation [22]	–	–	1
SC-Dataset [14]	Manually	–	8
SC-Dataset [10]	Manually	–	4
Vulnerable-VerifiedSCs [15]	35	–	10
SmarthER [21]	Slither	–	–
SCVHunter [16]	Manually	–	4
Eth-Reputable-Illicit-SC [20]	–	–	9
Contract-GNN [18]	Oyente	DASP	3
DAppSCAN [19]	Audit reports	SWC	34

Applications of Existing SC Datasets

Existing SC vulnerability datasets were employed in a variety of classification tasks. Some datasets supported binary classification models targeting specific vulnerability types. For example, the Dynamit dataset [24] was used to train classifiers that distinguish between benign and Reentrancy-related transactions based on behavioral features. Similarly, SC-Dataset [10] includes multiple vulnerability types but was constructed for binary classification, where each vulnerability is treated independently. This dataset was used for bytecode-based vulnerability detection [10]. It was also adopted to apply graph convolutional networks (GCNs) on opcode graphs for Reentrancy detection in decentralized finance (DeFi) SCs [34].

Other datasets provided broader vulnerability annotations. The Safe-SCs dataset [9] is a multi-class-labeled dataset originally proposed for sequence learning–based threat detection, where SC bytecode is analyzed to classify contracts as vulnerable or non-vulnerable. This dataset was also adapted into a two-stage classification pipeline that first identified vulnerable contracts and subsequently classified the vulnerability type [26].

Different datasets are designed to support multi-label classification [8,12,13], where a single SC is associated with multiple vulnerability types. These datasets are often reconstructed to support alternative classification tasks. For example, SoliAudit [8] targeted the DASP Top-10 SC vulnerabilities using opcode-level features and was evaluated under both binary and multi-label classification settings. It was also employed with hybrid deep learning architectures to classify contracts as vulnerable or non-vulnerable [28]. Similarly, ASSBert [27] reformulated vulnerability detection as a binary classification task and applied active and semi-supervised learning to reduce labeling effort.

SlitherAuditedContracts [12] is a multi-label dataset that supports multi-class classification tasks. It was used to construct an ensemble framework based on an enhanced genetic algorithm [30] for the classification of 11 vulnerability types. The same dataset was also reused under a binary classification setting, where transfer learning with a customized DistilBERT model was employed to detect Reentrancy vulnerabilities using deep contextualized code representations [31]. Similarly, ScrawlD [13] is a multi-label dataset that was adopted under a binary classification setting in a multimodal decision fusion approach [32] and in a detection framework based on the Model-Agnostic Meta-Learning (MAML) algorithm [33].

Existing datasets enabled significant advances in SC vulnerability detection. However, they were often designed to specific learning paradigms, vulnerability types, or detection stages, which limited their generalizability and reusability across diverse analysis tasks.

Gap Analysis

The review of existing literature highlights several limitations in publicly available SC vulnerability datasets. These gaps, along with the contributions of this study, are summarized as follows.

- Comprehensiveness and Coverage. Many datasets target specific compiler versions or narrow collection windows, which reduces generalizability and applicability. DIVE addresses this limitation by systematically acquiring SCs across versions and timeframes, thereby improving dataset representativeness.

- Labeling Accuracy and Standardization. Several datasets derive labels from only one or two analysis tools, which limits vulnerability coverage and introduces inaccuracies. When multiple tools are

combined, conflicts are often resolved through majority voting without validation. The lack of standardized taxonomies further reduces comparability and reuse. DIVE addresses these issues by integrating multiple analysis tools, employing a validated consensus method, and applying a standardized taxonomy to enhance labeling accuracy and ensure consistency across datasets.

- Feature Diversity and Automation. Few datasets provide a comprehensive set of SC attributes necessary for advanced security analysis. While source code enables extraction of features such as ASTs and code metrics, other attributes remain underexplored. DIVE's framework automates data acquisition and preprocessing, supports a wide range of features, and maintains a public feature registry that documents definitions and extraction procedures, promoting interpretability, reproducibility, and reuse.

- Dataset Availability and Sustainability. Well-structured and labeled datasets remain scarce, despite the public nature of blockchain data. The scale and heterogeneity of raw data, combined with the absence of complete pipelines, limit the creation of analysis-ready datasets. These challenges are compounded by the rapid evolution of SC ecosystems and the continuous emergence of vulnerabilities. This study addresses these issues by presenting DIVE, a multi-label and feature-rich dataset of SC vulnerabilities, and by releasing its supporting framework to enable researchers to extend existing datasets or construct new ones that reflect emerging trends.

Methods

The creation of the DIVE dataset is enabled by the publicly available DIVE framework[38], which was developed to automate and standardize the extraction and processing of SC data. The DIVE framework is implemented in Python 3.12.2 and integrates several open-source tools. It consists of three phases, as illustrated in Fig. 2. Each phase employs specialized modules for specific tasks, which are detailed in the following subsections.

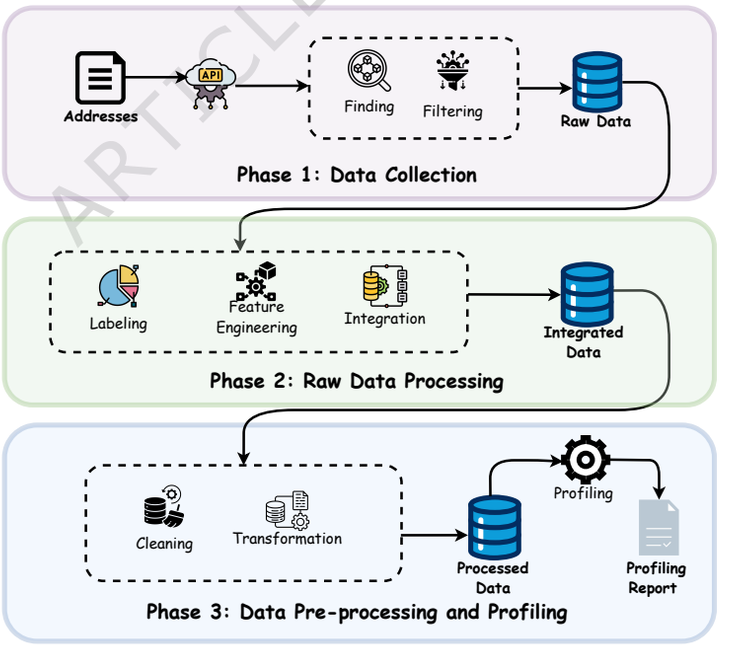


Fig. 2: Overview of the DIVE framework.

Data collection

Public blockchain data can be obtained from multiple sources, including direct access through blockchain nodes, open-source repositories, e.g., GitHub, and block explorers, e.g., Etherscan, that provide interfaces for querying on-chain data. The data collection phase is defined by two elements, scope and data type. Scope is specified by identifiers such as block numbers, timestamps, and contract addresses. The common blockchain data types are contract, transaction, and state data. This phase uses SC addresses to gather the following data.

- On-chain data: This involves retrieving essential data from contract deployment records using the contract address as an identifier.
- Off-chain data: Although SC source code is essential for vulnerability analysis, it is not stored on-chain. The framework retrieves the source code of verified contracts, together with associated code-based metadata, e.g., compiler version, optimization settings, license type, from Etherscan.

As illustrated in Fig. 3, this fundamental phase consists of three steps. It begins by reading a predefined list of SC addresses and proceeds to collect the raw data required for the subsequent phases. A detailed description of each step is as follows.

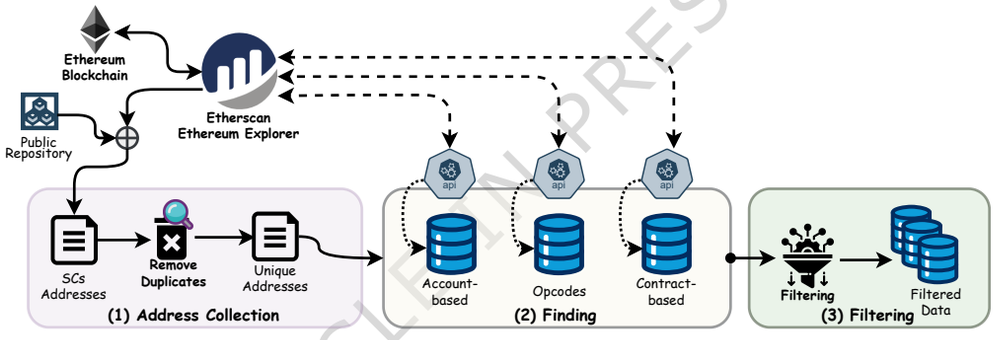


Fig. 3: Data collection phase.

1. **Address collection:** This step prepares a list of unique SC addresses for subsequent processing. For the DIVE dataset, addresses were collected from three sources: ScrawlD [13], Etherscan, and the SC Sanctuary repository [29]. The list is provided in CSV format. The *get_ Addresses* module ensures uniqueness and standardization by (1) removing duplicate entries and (2) renaming the address column to "contractAddress" for compatibility with other modules. To support maintainability, common label variations such as "contractaddress", "address", and "Addr" are predefined in the framework configuration file, allowing the module to automatically identify the address column in input files.
2. **Finding:** This step enables the retrieval of various Ethereum SC attributes using Etherscan APIs. The framework retrieves two categories of attributes: (1) code-based, including source code, bytecode, and opcodes, and (2) transaction-based attributes that describe contract characteristics. The *get_ ContractFeatures* module performs this retrieval using the first three Etherscan APIs listed in Table 4. This step produces 34 distinct attributes, each documented in the framework feature registry (`https://github.com/DIVE4Data/DIVE/blob/main/Featurelist.xlsx`).
3. **Filtering:** To ensure data integrity and consistency, the following basic preprocessing and filtering procedures are applied during this step.
    - Calling API 1 from Table 4 returns the ordered list of normal transactions associated with a given contract address, formally defined in Equation (1). Transaction-based features are extracted from the contract deployment transaction, i.e., the first transaction, which contains the creation bytecode and initialization attributes of the contract. In addition, the total

Table 4: Etherscan.io APIs.

API	Attributes	API String
1	Account-based	"https://api.etherscan.io/api?module=account&action=txlist&address={address}& apikey={api_key}"
2	Contract-based	"https://api.etherscan.io/api?module=contract&action=getsourcecode&address={address}& apikey={api_key}"
3	Opcodes	"https://api.etherscan.io/api?module=opcode&action=getopcode&address={address}& apikey={api_key}"
4	Transaction Count	"https://api.etherscan.io/api?module=proxy&action=eth_getBlockTransactionCountByNumber& tag={hex(blockNo)}& apikey={api_key}"

number of transactions associated with the address provides insights into contract usage and on-chain activity, therefore, this dynamic feature is also recorded (Equation (2)).

$$\mathcal{T}(addr) = (t_0, t_1, \ldots, t_{n_{addr}-1}), \qquad dt_{addr} = t_0 \tag{1}$$

$$n_{addr} = |\mathcal{T}(addr)| \tag{2}$$

where $addr$ denotes a SC address, $\mathcal{T}(addr)$ denotes the ordered list of transactions returned by the account-based API, dt_{addr} denotes the deployment transaction, and n_{addr} is the total number of transactions associated with $addr$.

- Calling API 2 from Table 4 returns contract metadata, including source code. Because source code is not directly utilized by data-driven applications, e.g., ML models, it is stored separately as a Solidity file. Solidity files can later be parsed to extract useful features, such as ASTs or code metrics, or to identify potential vulnerabilities.
- During data collection, Etherscan API calls may fail due to rate limits, unavailable metadata, or incomplete contract records. Contracts with missing or inconsistent critical attributes, e.g., absent source code, malformed bytecode, or incomplete deployment transactions, are removed to preserve dataset integrity.

Raw data processing

This phase prepares the data for analysis by applying several processes, as shown in Fig. 2, to ensure consistency and relevance. These processes address common challenges in data preparation, including labeling, feature engineering, and integration. Each process is described in detail as follows.

- **Data labeling:** Labeling of the DIVE dataset was performed in two steps, as follows.
  1. **Voting-based labeling:** A voting-based approach was applied using six analysis tools to label the DIVE dataset. The tools employed by Alsunaidi et al.[4] were used in this study, with configurations matching those reported in their work. Specifically, each contract was analyzed using MAIAN, Mythril, Semgrep, Slither, Solhint, and VeriSmart. The resulting reports were then processed by the MultiTagging framework[4], which parsed tool outputs and mapped tool-specific findings to standardized DASP Top 10 classes[39]. Final labels for each contract were assigned using the Power-based voting method[4], which determines the role of each tool and selects the optimal voting strategy for each vulnerability based on prior analytical evaluation.
  2. **Post-hoc validation:** Analysis tools are known to produce false positives[4,40,41]. Accordingly, a post-hoc validation stage is applied to reassess and reclassify positive findings when code evidence required for the respective DASP category is absent. This stage follows established practices for mitigating false positives reported in prior work[40,41]. The category-wise identification criteria are summarized in Table 5.

     The validation operates on the Power-based voting outputs generated by the MultiTagging framework and applies the rules defined in Table 5 through a gated decision mechanism. Each positive prediction is reassessed using the contract source code and tool-specific votes with their corresponding confidence scores. Findings that do not satisfy the vulnerability-specific criteria are reclassified as negative. The resulting assignments form the final revised labels of the DIVE dataset. For transparency and reproducibility, the validator implementation and configuration are publicly available[42].

Table 5: Post-hoc false positive identification criteria per vulnerability category.

Category	False Positive Identification Criteria
Reentrancy	The absence of an external call followed by a state update, or the presence of explicit reentrancy guards, e.g., `nonReentrant`.
Access Control	The presence of authorization checks, or the absence of privileged state updates that modify contract control or assets.
Arithmetic	Contracts compiled with Solidity version 0.8 or higher that lack `unchecked{}` blocks, or that employ the `SafeMath` library, where arithmetic overflow and underflow checks are enabled by default.
Unchecked Return Values	The absence of external calls, or the presence of explicit checks on return values from external calls or token transfers using `require` or equivalent assertions.
DoS	The absence of external calls within loops, the presence of statically bounded short loops, or functions declared as read-only (`view`/`pure`).
Bad Randomness	The absence of weak on-chain randomness sources, e.g., block attributes.
Front Running	The absence of externally reachable functions that modify economically sensitive state variables, or that rely on execution paths influenced by external transaction ordering.
Time Manipulation	Timestamps are not involved in control flow logic or comparison-based conditions.

- **Feature engineering:** This step extracts informative features from seven attributes to enhance dataset utility. It produces 214 derived features, documented in the feature registry, that provide deeper insights and may improve the performance of analytical applications. The following outlines the operations applied to each attribute.

  – **ABI-based features:** The ABI, typically provided in JSON format, defines the functions, events, and data types used in SC, enabling interaction between the contract and external systems. In this step, the ABI is parsed to identify the sets of functions and events and compute aggregate statistics over their inputs and outputs (Equations (3–6)). These elements are transformed into features that represent contract structure and behavior, yielding 19 additional features.

$$\begin{aligned} \mathcal{F}(addr) &= \{x \in \text{ABI}(addr) \mid \text{type}(x) = \texttt{function}\} \\ \mathcal{E}(addr) &= \{x \in \text{ABI}(addr) \mid \text{type}(x) = \texttt{event}\} \end{aligned} \tag{3}$$

$$\begin{aligned} &N_F = |\mathcal{F}(addr)|, \qquad N_E = |\mathcal{E}(addr)|, \\ &N_{\text{in}} = \sum_{f \in \mathcal{F}(addr)} |\text{In}(f)|, \qquad N_{\text{out}} = \sum_{f \in \mathcal{F}(addr)} |\text{Out}(f)| \end{aligned} \tag{4}$$

$$\begin{aligned} U_{\text{in}} &= \left| \bigcup_{f \in \mathcal{F}(addr)} \{\text{type}(p) \mid p \in \text{In}(f)\} \right| \\ U_{\text{out}} &= \left| \bigcup_{f \in \mathcal{F}(addr)} \{\text{type}(p) \mid p \in \text{Out}(f)\} \right| \end{aligned} \tag{5}$$

$$\overline{IO} = \frac{N_{\text{in}} + N_{\text{out}}}{N_F}, \qquad r_{\text{const}} = \frac{N_{\text{const}}}{N_F}, \qquad r_{\text{pay}} = \frac{N_{\text{pay}}}{N_F} \tag{6}$$

  where ABI($addr$) denotes the ABI JSON of contract $addr$, In(f) and Out(f) denote the input and output parameter lists of function f, type(p) denotes the Solidity data type of parameter p, $\overline{IO}$ denotes the average number of input and output parameters per function, N_{const} and N_{pay} denote the number of constant and payable functions in $\mathcal{F}(addr)$, respectively.

  – **Timestamp-based features:** This attribute captures temporal deployment characteristics. This step converts the Unix timestamp into a datetime object and extracts six features, including hour, weekday, quarter, and part of day, i.e., night, morning, afternoon, evening, as defined in Equation (9), using UTC. To reflect the cyclical nature of time, hour and weekday values are encoded using sine and cosine transformations (Equations (7 and 8)).

$$h = \text{hour}(\text{UTC}(t)), \qquad d = \text{dayofweek}(\text{UTC}(t)) \tag{7}$$

$$h_{\sin} = \sin\left(\frac{2\pi h}{24}\right), \quad h_{\cos} = \cos\left(\frac{2\pi h}{24}\right), \quad d_{\sin} = \sin\left(\frac{2\pi d}{7}\right), \quad d_{\cos} = \cos\left(\frac{2\pi d}{7}\right) \tag{8}$$

where t denotes the Unix timestamp, $h \in \{0, \ldots, 23\}$ and $d \in \{0, \ldots, 6\}$ denote the UTC hour and the day of the week, respectively.

$$\texttt{part\_of\_day}(h) = \begin{cases} \textit{night}, & 0 \leq h \leq 6 \\ \textit{morning}, & 7 \leq h \leq 12 \\ \textit{afternoon}, & 13 \leq h \leq 18 \\ \textit{evening}, & 19 \leq h \leq 23 \end{cases} \tag{9}$$

– **Library-based features:** This attribute captures external libraries linked to SC at deployment with their corresponding deployment addresses. This step extracts library names and counts, which may indicate modularity and dependency on external code. Such dependencies can increase attack surfaces or introduce vulnerabilities, especially when libraries are outdated or insecure.

– **TransactionIndex-based Features:** This attribute captures the position of the SC deployment transaction within its block. Because the number of transactions varies across blocks, using the raw `transactionIndex` may not provide meaningful comparative insight. To normalize this feature, the relative transaction position is computed as shown in Equation (10).

$$\text{Relative\_tx\_position} = \frac{\texttt{transactionIndex}}{\texttt{blockTransactionCount}} \tag{10}$$

where `blockTransactionCount` is retrieved via API 4 in Table 4.

This normalization enables consistent comparison across blocks of varying sizes. Based on the normalized value, the transaction is categorized into one of three positional classes, as defined in Equation (11).

$$\text{Block\_position} = \begin{cases} \textit{early}, & \text{if } \texttt{Relative\_tx\_position} \leq 0.25 \\ \textit{mid}, & \text{if } 0.25 < \texttt{Relative\_tx\_position} \leq 0.75 \\ \textit{late}, & \text{if } \texttt{Relative\_tx\_position} > 0.75 \end{cases} \tag{11}$$

– **Input-based features:** The `Input` attribute, retrieved from Etherscan, is a hex-encoded string that contains the contract's creation bytecode along with any constructor arguments. In this step, the hex-encoded creation bytecode is extracted and converted into raw bytes, which are then disassembled into opcodes defined by the Ethereum Virtual Machine (EVM) [43]. For example, the byte "0x60" corresponds to the opcode "PUSH1", which pushes one byte of data onto the stack. The resulting opcode sequence captures the contract's deployment logic. To ensure accuracy and consistency with the latest EVM specification, the framework automatically retrieves the opcode mapping from a publicly maintained registry [44].

– **Opcode-based features:** The opcode feature engineering process begins with tokenization, where the opcode sequence is split into individual instructions, or "tokens", e.g., "PUSH1", "ADDRESS", as defined in (Equation (12)). In the case of PUSH instructions, only the opcode itself is retained, while the immediate data that follows is excluded. The resulting tokens are categorized by functional role, such as arithmetic operations, environmental information, stack, memory, storage, or flow operations. From these representations, statistical features are derived, including opcode frequency and distribution (Equation (13)), the number and ratio of unique opcodes (Equation (14)), and category-wise opcode counts (Equation (15)). This process results in 153 derived features.

$$OP_{addr} = (op_1, op_2, \ldots, op_{L_{addr}}) \tag{12}$$

$$c_{addr}(op) = \sum_{k=1}^{L_{addr}} \mathbf{1}[op_k = op], \qquad f_{addr}(op) = \frac{c_{addr}(op)}{L_{addr}} \tag{13}$$

$$U_{addr} = |\{\, op_k \mid 1 \leq k \leq L_{addr} \,\}|, \qquad R_{addr} = \frac{U_{addr}}{L_{addr}} \tag{14}$$

$$C_{addr}(c) = \sum_{op \in \mathcal{V}_c} c_{addr}(op) \tag{15}$$

where *addr* denotes a SC address, OP_{addr} is the opcode sequence extracted from the contract bytecode, L_{addr} is the length of OP_{addr}, *op* denotes an EVM opcode, $c_{addr}(op)$ denotes the number of occurrences of opcode *op* in OP_{addr}, $\mathbf{1}[\cdot]$ is the indicator function, $\mathcal{V}_c$ denotes the set of opcodes belonging to category c, e.g., arithmetic, $f_{addr}(op)$ denotes the normalized opcode frequency, U_{addr} denotes the number of unique opcodes, R_{addr} denotes the unique opcode ratio, and $C_{addr}(c)$ denotes the category-wise opcode count.

– **Source code-based features (code metrics):** Code metrics capture structural properties and complexity, which are relevant for bug detection and maintainability analysis. As illustrated in Fig. 4, the framework employs *Solidity-Code-Metrics* (SCM) (`https://classic.yarnpkg.com/en/package/solidity-code-metrics`), an open-source tool that generates metrics in HTML or Markdown format. In this work, Markdown outputs are parsed using *Mrkdwn_analysis* (`https://pypi.org/project/markdown-analysis/`), a Python library, to extract metric-based features. The extracted data is then preprocessed for consistency, by removing duplicates, handling missing values, and encoding categorical variables. This process yields 31 features.

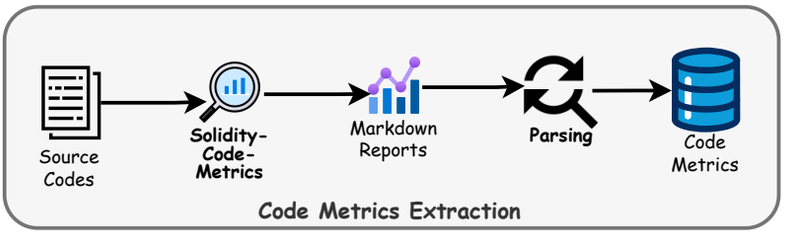


Fig. 4: Code metrics extraction.

- **Data integration:** This step integrates raw data, extracted features, and labels using the *construct_FinalData* module, resulting in a unified dataset. The module requires specification of the data sources and labels for integration. For the DIVE dataset, all collected attributes and labels are merged, then the *get_FilteredFeatures* method is applied to refine the dataset by enforcing filtering factors, such as pre-deployment and post-deployment stages, as defined in the feature registry.

Data preprocessing and profiling

This phase prepares the dataset for downstream tasks, such as building ML models or conducting other analyses. It consists of three procedures, implemented in *get_DataStatistics* modules, as follows.

- **Data cleaning:** This process ensures dataset integrity through two steps: (1) removing duplicates and (2) handling missing values. By default, missing values are replaced with zero, except where alternative values are more appropriate. Not all null values indicate missing information. For instance, in the *txreceipt_status* attribute, values of 0 or 1 indicate failure and success, respectively, while a missing value indicates a transaction awaiting confirmation. In such cases, missing values are replaced with -1 to denote a third state. This strategy ensures consistent handling of incomplete data. The dataset is also scanned to remove duplicate rows and features, preserving uniqueness and reliability for subsequent analyses.

- **Data transformation:** This step encodes categorical features using the *Label Encoder* technique (`https://scikit-learn.org/dev/modules/generated/sklearn.preprocessing.LabelEncoder.html`) from the *scikit-learn* library. Each category within a feature is assigned a unique integer, e.g., "red", "blue", and "green" become 0, 1, and 2. Boolean features are similarly encoded, e.g., "yes" = 1, "no" = 0. Hexadecimal features are converted into integers to ensure consistent numerical representations. All categorical, boolean, and hexadecimal features are explicitly defined in the framework configuration file to maintain accuracy and reproducibility.

- **Data profiling:** This step uses *ydata-profiling* v4.12.2 (`https://pypi.org/project/ydata-profiling/`), a tool that automates the generation of exploratory data analysis reports, to enhance dataset usability. These reports provide insights into data distributions, missing values, and potential outliers. The resulting profile establishes a baseline understanding of the dataset's structure and quality, supporting transparency and reliable analysis. Since *ydata-profiling* offers both minimal (fast) and detailed modes, the framework allows choosing the option that best balances performance and analytical depth.

In summary, the DIVE framework systematically progresses from SC addresses to a labeled dataset of vulnerabilities. While DIVE dataset creation followed a structured sequence, the framework's modular design provides flexibility, allowing procedures to be executed independently or reordered.

Data Records

The DIVE dataset is publicly available on Zenodo[25]. All records are distributed in machine-readable formats (CSV, xlsx, sol, and JSON) to ensure accessibility and interoperability. Table 6 provides an overview of the repository structure. The repository comprises three principal components: raw API outputs, cleaned and preprocessed datasets, and contract labels.

To promote transparency and reproducibility, a detailed machine-readable feature registry is included, documenting all extracted features and their associated metadata. Additionally, exploratory data analysis (EDA) and profiling reports are provided to facilitate dataset understanding and reuse.

Table 6: Overview of files in the DIVE repository.

File Name	Description
DIVE_Raw_Data.zip	Contains the unprocessed attributes corresponding to both the pre-deployment (PRE) and post-deployment (POST) stages of the SC lifecycle.
DIVE_Preprocessed_Data.zip	Includes the files PRE_ProcessedData.csv and POST_ProcessedData.csv, which are prepared for ML tasks and analytical applications.
DIVE_Labels.zip	Provides the DIVE labeling files, namely Tool_Results.csv and DIVE_Labels.csv.
Feature list.xlsx	Provides a comprehensive feature catalog detailing each feature's description, relevance, category, status (static or dynamic), feature and data types, preprocessing requirements, extraction and collection sources, and availability.
EDA_and_Profiling_Reports.zip	Contains EDA results, data profiling reports, and feature distribution visualizations for both the PRE and POST datasets.

To enhance usability, the dataset is structured along two complementary dimensions. First, the dataset is organized into raw and preprocessed versions, enabling researchers to either begin from the original on-chain/off-chain representations or directly employ curated features. Second, the dataset is divided into pre-deployment and post-deployment layers, reflecting the natural lifecycle of SCs. This section provides a detailed description of both raw and preprocessed features across these lifecycle layers, as well as the data labels.

Pre-Deployment Data

The pre-deployment data represent intrinsic contract properties available prior to deployment on the blockchain. Table 7 presents the raw attributes retrieved using Etherscan APIs. These attributes were processed and normalized to produce a feature set suitable for ML analysis. The final feature set includes:

- Five code-based features: *CompilerVersion, OptimizationUsed, Runs, EVMVersion, LicenseType.*
- Two features describing external libraries: *ExternalLibNames, NoOfExternalLib.*
- 15 ABI-derived metrics capturing structural properties such as the number and types of functions and events.
- 31 source code metrics reflecting syntactic and complexity characteristics of Solidity codes.

- 168 opcode-based metrics summarizing instruction frequency and control-flow patterns in creation bytecode.

Table 7: Pre-deployment raw attributes.

Attribute	Description	File
Code metadata	Includes *ABI, ContractName, CompilerVersion, OptimizationUsed, Runs, ConstructorArguments, EVMVersion, Library, LicenseType, SwarmSource*	Code-based.csv
Source code	Solidity source code for each SC	.sol files
Input	Creation bytecode and constructor arguments used to generate the runtime code	Input.jsonl
Creation opcodes	Disassembled opcode sequence from creation bytecode	Creation_Opcode.jsonl

These features capture intrinsic aspects of SCs available before deployment. Code metadata reflects compilation choices and licensing information, library references indicate external dependencies, ABI metrics describe contract interfaces, source code metrics quantify structural and syntactic complexity, and opcode metrics summarize instruction patterns in creation bytecode.

Post-Deployment Data

The post-deployment data represent extrinsic contract properties observable on-chain after compilation and deployment. Table 8 presents the raw attributes retrieved via Etherscan APIs. These attributes were processed and normalized to create features suitable for ML analysis. The final feature set includes:

- Ten features from contract account and deployment transaction metadata: *NoOfTransactions, nonce, value, gas, gasPrice, txreceipt_status, cumulativeGasUsed, gasUsed, methodId, confirmations.*
- Six temporal features derived from timestamp: *hour_sin, hour_cos, dayofweek_sin, dayofweek_cos, quarter, part_of_day.*
- Two features from the transactionIndex representing intra-block ordering: *relative_tx_position* (numerical) and *block_position* (categorical).
- 158 opcode-based metrics summarizing runtime instruction usage and execution semantics

Table 8: Post-deployment raw attributes.

Attribute	Description	File
Deployment transaction metadata	Includes *NoOfTransactions, blockNumber, timestamp, hash, nonce, blockHash, transactionIndex, from, to, value, gas, gasPrice, isError, txreceipt_status, cumulativeGasUsed, gasUsed, confirmations, methodId, functionName, Proxy, Implementation*	Transaction-based.csv
Runtime Opcode	Opcode sequence for each SC, generated by disassembling the runtime bytecode	Runtime_Opcode.jsonl

These features capture complementary aspects of deployed SCs. Transaction metadata reflects deployment behavior and resource usage, temporal features encode cyclic and periodic trends, transaction position features highlight intra-block ordering effects, and opcode metrics summarize execution level semantics relevant to vulnerability detection.

Labels

The dataset includes vulnerability labels aligned with the DASP Top 10 taxonomy. Eight categories are represented, namely Reentrancy, Access control, Arithmetic issues, Unchecked low-level calls, DoS, Bad randomness, Front running, and Time manipulation. Each contract is annotated using a multi-label scheme, as a single SC may exhibit multiple vulnerability types.

Labels are provided in two forms, as shown in Table 9. The first form consists of individual tool labels, where each tool exposes eight binary columns. A value of 1 indicates that the tool flagged the corresponding vulnerability pattern, whereas 0 indicates no finding. The second form consists of the final DIVE dataset labels, which contain aggregated labels provided by the Power-based voting method [4], followed by verification and filtering to reduce label noise. The inclusion of individual tool predictions enables researchers to explore alternative aggregation strategies, including majority voting and weighted voting, beyond the provided final labels.

Table 9: Labels format of the DIVE dataset.

Format	**Description**	**File**
Tool labels	Binary labels produced by each tool	Tool_Results.csv
Aggregated labels	Final labels after aggregation, verification, and filtering	DIVE_Labels.csv

Data Overview

During data collection, contract metadata was obtained using three independent Etherscan APIs. As shown in Table 10, an initial set of 32,766 SC addresses, published in the DIVE framework repository [38], was queried to construct the DIVE dataset. To ensure feature completeness, only contracts for which all three APIs returned data were retained, yielding 23,228 contracts. Among these, 22,330 contracts were analyzed by at least one SC analysis tool to derive dataset labels and were included in the final dataset. Contracts not analyzed by any tool were excluded, as no labels could be derived.

Table 10: Statistics on DIVE data collection and filtering.

Data collection stage	**API 1**	**API 2**	**API 3**
Queried contract addresses	32,766	32,766	32,766
Retrieved samples per API	23,354	25,563	32,766
Missing API data	9,412	7,203	0
Addresses with complete API data (intersection)		23,228	
Retained addresses (labeled samples)		22,330	

The final dataset comprises 22,330 verified Ethereum SCs retrieved from Etherscan. Fig. 5a shows that the contract deployment years range from 2016 to 2024. The number of verified contracts is relatively low in some years, which affects the overall distribution of samples. The dataset further covers a wide range of Solidity compiler versions, from v0.4.x to v0.8.x. As shown in Fig. 5b, version v0.8.x is the most recent and widely adopted, whereas versions v0.5.x, v0.6.x, and v0.7.x are underrepresented, reflecting their limited use among verified contracts on Etherscan.

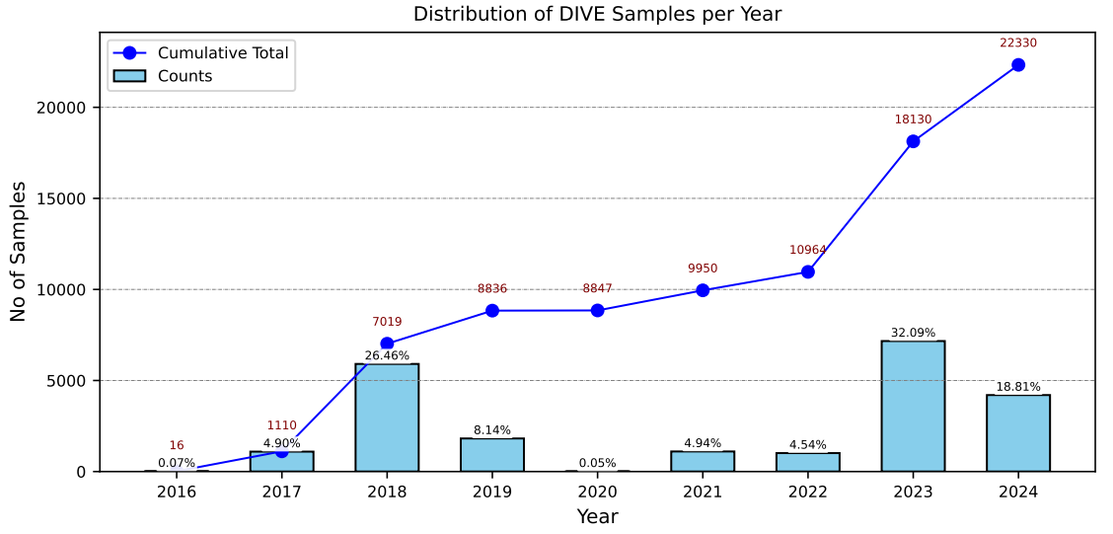
As shown in Fig. 6, the distribution is imbalanced, categories such as Access control and Reentrancy occur more frequently than Bad randomness and Front running. This imbalance may hinder ML training and evaluation. Researchers can mitigate it through resampling strategies, data augmentation, or loss functions that assign greater weight to minority classes to achieve more balanced learning outcomes [39].

Fig. 7a presents vulnerability label co-occurrence probabilities computed using the conditional probability [45] defined in Equation (16). Each matrix entry denotes the probability of observing label j given label i, with columns corresponding to j and rows to i. Values fall within the range $[0, 1]$, with higher values indicating stronger co-occurrence.

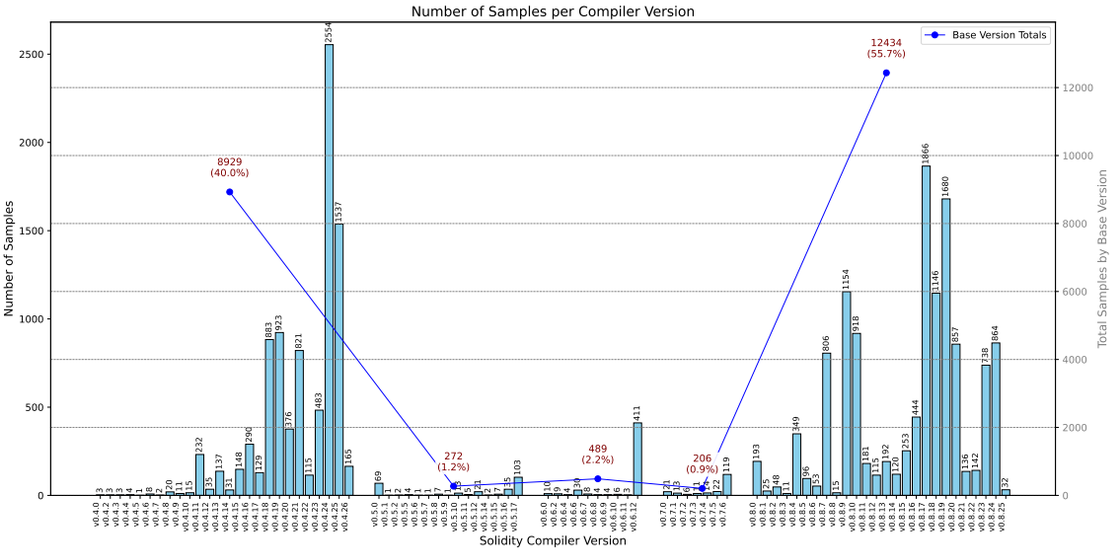
$$P(j \mid i) = \frac{\sum_{n=1}^{N} Y_{n,i}\, Y_{n,j}}{\sum_{n=1}^{N} Y_{n,i}}, \tag{16}$$

where $Y_{n,i}, Y_{n,j} \in \{0, 1\}$ denote label indicators for sample n, and N is the number of samples.

The figure reveals that several vulnerabilities frequently co-occur, indicating non-independent label distributions. In particular, Reentrancy and Access control show consistently high co-occurrence with multiple vulnerability types, whereas vulnerabilities such as Bad randomness and Front running appear



(a) Distribution of DIVE samples per year.



(b) Number of samples per Solidity compiler version.

Fig. 5: Distribution of samples in the DIVE dataset.

more selectively. These patterns highlight structured dependencies among labels, motivating the use of multi-label and dependency-aware learning approaches.

To further assess label dependency, lift is computed as defined in Equation (17). Lift is a standard association-based measure expressed as the ratio between the conditional probability and the marginal probability of the consequent [46]. Lift values greater than one indicate positive dependency, values equal to one indicate independence, and values below one indicate negative dependency, indicating that the presence of one vulnerability reduces the likelihood of the other.

$$\text{Lift}(i, j) = \frac{P(j \mid i)}{P(j)}, \tag{17}$$

where $P(j) = \frac{1}{N} \sum_{n=1}^{N} Y_{n,j}$ denotes the marginal probability of label j.

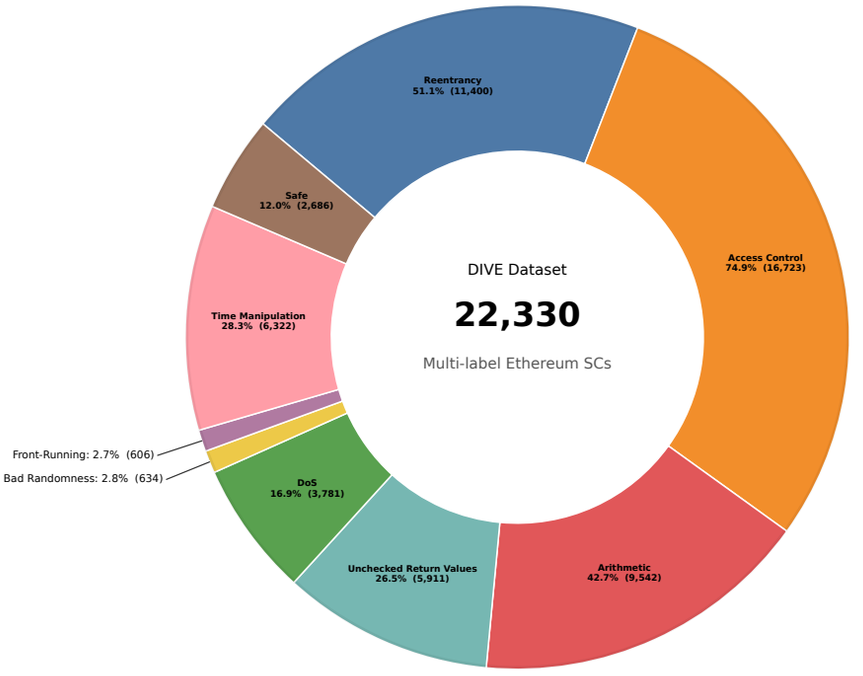


Fig. 6: Distribution of labels in the DIVE dataset.

Fig. 7b shows several strong dependencies between vulnerabilities, most notably between Front running and Bad randomness, as well as between Front running and DoS.

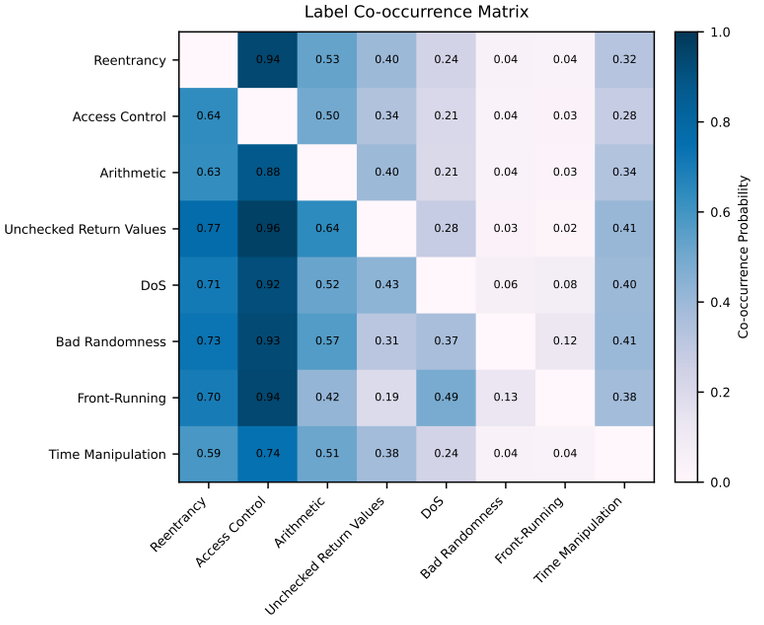
Technical Validation

This work evaluates six critical data quality attributes, five of which align with the ISO/IEC 25012 standard[47]. Croft et al.[48] emphasize the importance of uniqueness in ML datasets, since duplicate instances can bias models and reduce generalizability. The following discusses each attribute and how it is addressed in the DIVE dataset.

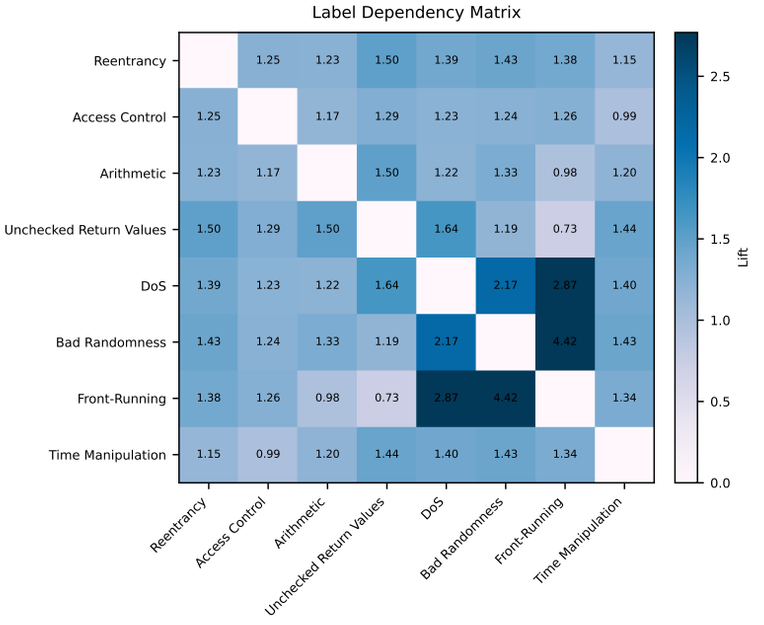
1. Accuracy: Accuracy covers both contract-level data and vulnerability labels and is addressed through three validation measures. First, each instance is validated through automated checks within the DIVE framework to ensure inclusion of only contract samples associated with a deployment transaction, i.e., a transaction with an empty "To" field, and verified source code. Second, labels are assigned using the Power-based voting method, which aggregates outputs from six established analysis tools, as described in MultiTagging[4], and are subsequently refined through post-hoc filtering rules. This step reduces false positives introduced by the analysis tools based on vulnerability-specific code evidence summarized in Table 5. Although this approach reduces false positives, it introduces an additional heuristic layer that should be considered when interpreting labeling outcomes. Third, automated preprocessing in DIVE ensures consistent normalization and enhances dataset interpretability.

2. Completeness: The dataset achieves completeness by covering verified Ethereum SCs deployed from 2016 to 2024 (Fig. 5a) and compiler versions v0.4.x–v0.8.x (Fig. 5b). Labels cover eight categories of the DASP Top 10[4], excluding the last two: "Short address attack" and "Unknown unknowns". The former involves external manipulation of transaction data and does not directly reflect contract-level code security[49].

   Within the DASP taxonomy, the category "Unknown unknowns" refers to vulnerability classes that are not yet formally characterized or lack consistent detection patterns. It functions as a placeholder for emerging or unforeseen threats rather than a concrete and operationally defined vulnerability type. Due to its inherently undefined scope and absence of reproducible labeling



(a) Conditional co-occurrence of vulnerability labels.



(b) Label dependency measured using lift.

Fig. 7: Co-occurrence and dependency patterns of vulnerability labels in the DIVE dataset.

criteria, this category is excluded from DIVE to preserve annotation consistency and methodological transparency.

The dataset further provides raw artifacts, including source code, bytecode, opcodes, and deployment transaction attributes, along with a processed tabular representation containing more than 200 features spanning pre- and post-deployment aspects, with automated handling of missing values.

3. Consistency: The dataset consistency is ensured by standardizing all feature formats, including

transaction metadata, to align with common standards in similar datasets. During the data preprocessing phase of the DIVE framework, any format inconsistencies, such as those in compiler versions, were automatically resolved. This systematic approach guarantees uniformity across all entries, enhancing the reliability and consistency of data analysis.

4. Credibility: This attribute is established through systematic data collection and labeling with verifiable provenance. Raw contract data were retrieved from Etherscan, an authoritative source for Ethereum, and preserved using contract addresses. Labels were generated with the MultiTagging framework[4] under documented tool versions and configurations, and were subsequently subjected to post-hoc validation to mitigate false positives. Post-hoc filtering detected and corrected false positives across vulnerability categories, with correction rates of 0.7% for both Access control and Arithmetic, 14.3% for DoS, and 24.9% for Time manipulation, while no corrections were observed for the remaining vulnerability classes. The complete framework, from data collection to preprocessing, is publicly released to support reproducibility and reuse.

5. Currentness: The dataset includes samples up to 2024, capturing recent SC vulnerabilities and development practices. The publicly available DIVE framework allows researchers to extend the dataset with new contract samples, ensuring its continued timeliness.

6. Uniqueness: Duplicate entries are systematically removed during the preprocessing phase to eliminate redundant contract records. However, different contracts may still share identical opcode sequences. To quantify structural similarity, the uniqueness factor is defined in Equation (18) as the proportion of contracts whose normalized opcode sequence appears exactly once in the dataset.

$$UniquenessFactor = \frac{N_{\text{unique}}}{N_{\text{total}}} \times 100\% \tag{18}$$

where N_{total} is the total number of contracts analyzed and N_{unique} is the number of contracts with unique opcode sequences.

The uniqueness factor is computed over normalized opcode sequences derived from contract bytecode. Opcode strings are tokenized and filtered to retain mnemonic tokens, which are converted to lowercase, and immediate operands of PUSH instructions are excluded. Only opcode mnemonics are retained, and PUSH instructions are treated uniformly irrespective of the literal values they contain. The resulting normalized opcode sequence is hashed to identify contracts that share identical execution templates. This normalization ensures that the uniqueness measure captures structural execution patterns rather than superficial differences in embedded constants or formatting.

For the DIVE dataset of 22,330 SCs, 18,875 (84.53%) exhibit unique opcode sequences, while 3,455 (15.47%) share their opcode skeleton with at least one other contract. The largest duplicate family contains 241 SCs, and the ten most common opcode templates account for 3.30% of the dataset. This degree of duplication raises important considerations for ML applications. When only opcode sequences are used as features, models are prone to overfitting to dominant patterns, leading to overestimated performance and reduced generalizability.

Limitations

This section discusses limitations associated with both the DIVE framework and the resulting dataset.

Framework-Level Limitations

The current DIVE framework targets the Ethereum blockchain and relies primarily on Etherscan as its data source. It focuses on code-based artifacts from verified SCs and their deployment transactions. As a result, other contract-related data types, such as detailed transaction histories and contract interaction graphs, are not currently supported. These constraints reflect design choices made to ensure data reliability and reproducibility, and they may limit the applicability of the framework to broader blockchain ecosystems.

DIVE adopts a modular design and can be extended in future work to support additional blockchain platforms, incorporate richer data types, and integrate advanced feature engineering techniques. Such extensions could also enable automated dataset quality monitoring mechanisms, including concept and

data drift detection. These enhancements have the potential to broaden the applicability of DIVE as a platform for the development and validation of vulnerable SC datasets.

Dataset-Level Limitations

The DIVE dataset exhibits the following limitations that require consideration.

- **Label Quality and Annotation Noise.** Vulnerability labels are generated through a multi-stage automated pipeline based on static and symbolic SC analysis tools. DIVE combines six established analyzers using the Power-based voting method[4], followed by rule-based post-hoc validation that reassesses positive findings against vulnerability-specific code evidence.

  This design reduces known sources of spurious detections, e.g., SafeMath-protected arithmetic or reentrancy warnings without state modification. Nevertheless, the annotations remain dependent on the capabilities and limitations of automated analyzers. Static and symbolic tools may produce false positives and false negatives, particularly for vulnerabilities requiring execution context, environmental assumptions, cross-contract interactions, or complex state evolution. Accordingly, the labels should be interpreted as systematically derived, high-confidence annotations rather than manually verified ground truth.

  Due to the scale of the dataset, manual audits and validation against external ground-truth annotations or audit reports were not conducted. Instead, label reliability is supported through tool diversity, calibrated voting, and explicit validation rules, which are publicly released to ensure transparency and reproducibility. This release does not include formal empirical estimates of precision or recall. DIVE is therefore appropriate for comparative evaluation, representation learning, and robustness analysis. Incorporating targeted manual validation and benchmark-aligned auditing represents an important direction for future extensions of the dataset.

- **Dataset Bias and Representativeness.** DIVE consists exclusively of verified Ethereum SCs obtained from Etherscan, where verification indicates that the published source code matches the deployed bytecode. This requirement enables source code–based feature extraction and the construction of multiple SC representations. However, verification does not imply security, correctness, or audit guarantees. Verified contracts may still contain vulnerabilities or unsafe behaviors. For instance, the contract at address "`0x40335f6c7503f588a6c8f2c86d446550ab18b750`" is verified on Etherscan, yet its source code triggers multiple Solidity compiler warnings related to known low- and medium-severity issues.

  Restricting data collection to verified contracts yields a subset of the Ethereum ecosystem that does not necessarily represent all deployed contracts. Verified contracts are more likely to be developer-published and documented and may therefore underrepresent bytecode-only, intentionally obfuscated, short-lived, illicit, or adversarial contracts commonly encountered in real-world exploitation scenarios. Consequently, certain behaviors and contract types are likely underrepresented. Models trained exclusively on DIVE may therefore not fully capture characteristics of malicious or deliberately evasive contracts, and caution is warranted when generalizing findings to adversarial or heterogeneous deployment environments.

- **Class Imbalance and Distribution Skew.** As illustrated in Fig. 5 and 6, DIVE exhibits significant skew across both deployment years and vulnerability categories. Certain classes, e.g., Access control and Reentrancy, dominate the dataset, whereas others, e.g., Bad randomness and Front running, appear relatively infrequently. This distribution may reflect differences in real-world prevalence and tool-specific detection characteristics. However, it may also introduce bias in trained ML models and limit generalization across vulnerability types, particularly in settings that require balanced performance across categories. Addressing this imbalance typically requires the use of imbalance-aware learning strategies[39], including resampling techniques, class weighting, cost-sensitive training, and class-aware evaluation protocols during model development and evaluation.

- **Scope and Coverage.** The dataset represents the state of SCs at the time of collection and does not reflect subsequent updates or patches, which may affect its long-term relevance. As the DIVE framework used to construct the dataset is publicly available, this limitation can be mitigated through periodic dataset regeneration to account for data and concept drift.

Future work may focus on enriching the dataset with new attributes, including transaction records and contract interactions, assessing its performance across a broader range of ML models, and examining how different feature types influence SC vulnerability detection.

Usage Notes

Several considerations arise when analyzing and interpreting results from applied or experimental studies using the DIVE dataset. Owing to its multi-label structure and class imbalance, evaluation metrics such as the macro F1 score and the Area Under the Precision–Recall Curve (PR-AUC) are recommended. Unlike accuracy, these metrics assign equal weight to all categories and are less affected by class dominance. Furthermore, per-class performance analysis is advisable to prevent conclusions driven primarily by majority categories, such as Reentrancy and Access control.

As discussed in the Data Overview section, vulnerability labels in the DIVE dataset demonstrate significant co-occurrence and dependency patterns. These relationships indicate that labels are not independent and should be considered when designing and evaluating multi-label or multi-task learning models. Approaches that account for label dependencies may therefore be more suitable than methods that treat each label independently.

The dataset includes features representing two stages of the SC lifecycle, enabling comparative analyses across pre-deployment and post-deployment settings. Models trained on code-based features reflect the ability to detect vulnerabilities introduced during the contract development phase, whereas models using post-deployment features can detect vulnerabilities that emerge during execution. Differences in performance across these settings can provide insight into the temporal detectability of vulnerabilities and constraints arising from stage-dependent feature availability, as certain runtime features are only observable after deployment, while source code may not always be accessible post-deployment.

Data uniqueness should be verified in experiments that rely exclusively on opcode-based features. As discussed in the Technical Validation section, a subset of SCs shares identical opcode structures. This level of duplication can lead to data leakage between training and test splits, resulting in overestimated performance and reduced generalizability if not properly controlled.

Labels in DIVE are aggregated using the Power-based voting method [4], followed by post hoc validation to correct false labels. Labels generated by individual tools are also provided to enable the evaluation of alternative aggregation strategies or comparison across analytical tools. When applying a different aggregation method, label validation is recommended to mitigate errors introduced by individual tools.

Data Availability

The dataset generated using the DIVE framework is publicly available on the Zenodo [25] under the Creative Commons Attribution 4.0 International (CC BY 4.0) license. The repository includes the dataset, the associated metadata registry, and data profiling information to support understanding and reuse.

Code Availability

The open-source DIVE framework is available on Zenodo [38] under the Creative Commons Attribution–NonCommercial 4.0 International (CC BY-NC 4.0) license. The repository includes documentation and usage instructions.

References

1. Bocek, T. & Stiller, B. Smart contracts–blockchains in the wings. In Linnhoff-Popien, C., Schneider, R. & Zaddach, M. (eds.) *Digital Marketplaces Unleashed*, 169–184 (Springer, Berlin, 2018). `https://doi.org/10.1007/978-3-662-49275-8_19`.

2. Zheng, Z. *et al.* An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems* **105**, 475–491 (2020). `https://doi.org/10.1016/j.future.2019.12.019`.

3. Kushwaha, S. S., Joshi, S., Singh, D., Kaur, M. & Lee, H.-N. Systematic review of security vulnerabilities in ethereum blockchain smart contract. *IEEE Access* **10**, 6605–6621 (2022). https://doi.org/10.1109/ACCESS.2021.3140091.

4. Alsunaidi, S. J., Aljamaan, H. & Hammoudeh, M. MultiTagging: A vulnerable smart contract labeling and evaluation framework. *Electronics* **13**, 4616 (2024). https://doi.org/10.3390/electronics13234616.

5. Ivanov, N. *et al.* Security threat mitigation for smart contracts: A comprehensive survey. *ACM Computing Surveys* **55**, 1–37 (2023). https://doi.org/10.1145/3593293.

6. Jiang, F. *et al.* Enhancing smart-contract security through machine learning: A survey of approaches and techniques. *Electronics* **12**, 2046 (2023). https://doi.org/10.3390/electronics12092046.

7. Alsunaidi, S. J. & Alhaidari, F. A. A survey of consensus algorithms for blockchain technology. In *2019 International Conference on Computer and Information Sciences (ICCIS)*, 1–6 (IEEE, 2019). https://doi.org/10.1109/ICCISci.2019.8716424.

8. Liao, J.-W., Tsai, T.-T., He, C.-K. & Tien, C.-W. Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, 458–465 (IEEE, 2019). https://doi.org/10.1109/IOTSMS48152.2019.8939256.

9. Tann, W. J.-W., Han, X. J., Gupta, S. S. & Ong, Y.-S. Towards safer smart contracts: A sequence learning approach to detecting security threats. Preprint at https://doi.org/10.48550/arXiv.1811.06632 (2018).

10. Qian, P., Liu, Z., Yin, Y. & He, Q. Cross-modality mutual learning for enhancing smart contract vulnerability detection on bytecode. In *Proceedings of the ACM Web Conference 2023*, 2220–2229 (2023). https://doi.org/10.1145/3543507.3583367.

11. Zhang, L. *et al.* SPCBIG-EC: a robust serial hybrid model for smart contract vulnerability detection. *Sensors* **22**, 4621 (2022). https://doi.org/10.3390/s22124621.

12. Rossini, M. Slither audited smart contracts dataset. *Hugging Face* https://huggingface.co/datasets/mwritescode/slither-audited-smart-contracts (2022).

13. Yashavant, C. S., Kumar, S. & Karkare, A. Scrawld: A dataset of real world ethereum smart contracts labelled with vulnerabilities. Preprint at https://doi.org/10.48550/arXiv.2202.11409 (2022).

14. Liu, Z. *et al.* Rethinking smart contract fuzzing: Fuzzing with invocation ordering and important branch revisiting. *IEEE Transactions on Information Forensics and Security* **18**, 1237–1251 (2023). https://doi.org/10.1109/TIFS.2023.3237370.

15. Storhaug, A. Vulnerable verified smart contracts (v2). *figshare* https://doi.org/10.6084/m9.figshare.21990287.v2 (2023).

16. Luo, F. *et al.* Scvhunter: Smart contract vulnerability detection based on heterogeneous graph attention network. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13 (2024). https://doi.org/10.1145/3597503.3639213.

17. Durieux, T., Ferreira, J. F., Abreu, R. & Cruz, P. 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 (2020). https://doi.org/10.1145/3377811.3380364.

18. Wang, Y., Zhao, X., He, L., Zhen, Z. & Chen, H. ContractGNN: Ethereum smart contract vulnerability detection based on vulnerability sub-graphs and graph neural networks. *IEEE Transactions on Network Science and Engineering* **11**, 6382–6395 (2024). https://doi.org/10.1109/TNSE.2024.3470788.

19. Zheng, Z. *et al.* DAppSCAN: Building large-scale datasets for smart contract weaknesses in dapp projects. *IEEE Transactions on Software Engineering* **50**, 1360–1373 (2024). https://doi.org/10.1109/TSE.2024.3383422.

20. Malik, C. eth-reputable-illicit-sc-code. *Hugging Face* https://huggingface.co/datasets/malikcyrus/eth-reputable-illicit-sc-code (2024).

21. Ibba, G. *et al.* A curated solidity smart contracts repository of metrics and vulnerability. In *Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering*, 32–41 (2024). https://doi.org/10.1145/3663533.3664039.

22. Forta. token-impersonation-dataset. *Hugging Face* https://huggingface.co/datasets/forta/token-impersonation-dataset (2023).

23. Ajienka, N. Supervised machine learning for smart contract vulnerability prediction (v2). *figshare* https://doi.org/10.6084/m9.figshare.13417316.v2 (2020).

24. Eshghie, M., Artho, C. & Gurov, D. Dynamic vulnerability detection on smart contracts using machine learning. In *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, 305–312 (2021). https://doi.org/10.1145/3463274.3463348.

25. Alsunaidi, S., Aljamaan, H. & Hammoudeh, M. DIVE: A multi-label smart contract vulnerability dataset. *Zenodo* https://doi.org/10.5281/zenodo.18519253 (2026).

26. Mezina, A. & Ometov, A. Detecting smart contract vulnerabilities with combined binary and multi-class classification. *Cryptography* **7**, 34 (2023). https://doi.org/10.3390/cryptography7030034.

27. Sun, X. *et al.* ASSBert: Active and semi-supervised bert for smart contract vulnerability detection. *Journal of Information Security and Applications* **73**, 103423 (2023). https://doi.org/10.1016/j.jisa.2023.103423.

28. Jain, V. K. & Tripathi, M. An integrated deep learning model for ethereum smart contract vulnerability detection. *International Journal of Information Security* **23**, 557–575 (2024). https://doi.org/10.1007/s10207-023-00752-5.

29. Ortner, M. & Eskandari, S. Smart contract sanctuary–Ethereum. *GitHub* https://github.com/tintinweb/smart-contract-sanctuary-ethereum (2023).

30. HajiHosseinKhani, S., Lashkari, A. H. & Oskui, A. M. Unveiling smart contracts vulnerabilities: Toward profiling smart contracts vulnerabilities using enhanced genetic algorithm and generating benchmark dataset. *Blockchain: Research and Applications* **6**, 100253 (2024). https://doi.org/10.1016/j.bcra.2024.100253.

31. Lê Hùng, B. *et al.* Contextual language model and transfer learning for reentrancy vulnerability detection in smart contracts. In *Proceedings of the 12th International Symposium on Information and Communication Technology*, 739–745 (ACM, New York, NY, USA, 2023). https://doi.org/10.1145/3628797.3628945.

32. Deng, W. *et al.* Smart contract vulnerability detection based on deep learning and multimodal decision fusion. *Sensors* **23**, 7246 (2023). https://doi.org/10.3390/s23167246.

33. Yang, Z., Zhu, W. & Yu, M. Improvement and optimization of vulnerability detection methods for ethernet smart contracts. *IEEE Access* **11**, 78207–78223 (2023). https://doi.org/10.1109/ACCESS.2023.3298672.

34. Lakadawala, H., Dzigbede, K. & Chen, Y. Detecting reentrancy vulnerability in smart contracts using graph convolution networks. In *2024 IEEE 21st Consumer Communications & Networking Conference (CCNC)*, 188–193 (IEEE, Las Vegas, NV, USA, 2024). https://doi.org/10.1109/CCNC51664.2024.10454763.

35. Hu, T. A benchmark dataset of solidity smart contracts. *Zenodo* http://dx.doi.org/10.5281/zenodo.7744053 (2023).

36. Ghaleb, A. & Pattabiraman, K. 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 (2020). https://doi.org/10.1145/3395363.3397385.

37. Zhou, H., Milani Fard, A. & Makanju, A. The state of ethereum smart contracts security: Vulnerabilities, countermeasures, and tool support. *Journal of Cybersecurity and Privacy* **2**, 358–378 (2022). https://doi.org/10.3390/jcp2020019.

38. DIVE Framework. DIVE (version v2.0.0). *Zenodo* https://doi.org/10.5281/zenodo.18779606 (2026).

39. Alsunaidi, S. J., Aljamaan, H. & Hammoudeh, M. Leveraging machine learning models to improve smart contract security: A survey of vulnerabilities and detection methods. *ACM Computing Surveys* **58**, 1–37 (2025). https://doi.org/10.1145/3772367.

40. Yu, R., Shu, J., Yan, D. & Jia, X. Redetect: Reentrancy vulnerability detection in smart contracts with high accuracy. In *2021 17th International Conference on Mobility, Sensing and Networking (MSN)*, 412–419 (IEEE, 2021). https://doi.org/10.1109/MSN53354.2021.00069.

41. Zheng, Z. *et al.* Turn the rudder: A beacon of reentrancy detection for smart contracts on ethereum. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 295–306 (IEEE, 2023). https://doi.org/10.1109/ICSE48619.2023.00036.

42. Label post-hoc validator. *GitHub* https://github.com/DIVE4Data/Label-Post-hoc-Validator (2026).

43. Wood, G. Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper https://mholende.win.tue.nl/seminar/references/ethereum_yellowpaper.pdf (2014).

44. Ethereum Foundation. Ethereum virtual machine (EVM) opcodes documentation (2025). https://ethereum.org/en/developers/docs/evm/opcodes/.

45. Rawlekar, S., Bhatnagar, S., Srinivasulu, V. P. & Ahuja, N. Improving multi-label recognition using class co-occurrence probabilities. In *International Conference on Pattern Recognition*, 424–439 (Springer, 2024). https://doi.org/10.1007/978-3-031-78192-6_28.

46. Grabot, B. Rule mining in maintenance: Analysing large knowledge bases. *Computers & Industrial Engineering* **139**, 105501 (2020). https://doi.org/10.1016/j.cie.2018.11.011.

47. ISO/IEC. ISO/IEC 25012: Software engineering—software product quality requirements and evaluation (SQuaRE)—data quality model. International Standard https://www.iso.org/obp/ui/#iso:std:iso-iec:25012:ed-1:v1:en (2008).

48. Croft, R., Babar, M. A. & Kholoosi, M. M. Data quality for software vulnerability datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 121–133 (IEEE, 2023). https://doi.org/10.1109/ICSE48619.2023.00022.

49. Bylica, P. How to find $10m just by reading the blockchain. *Medium* https://medium.com/golem-project/how-to-find-10m-by-just-reading-blockchain-6ae9d39fcd95 (2017).

Acknowledgements

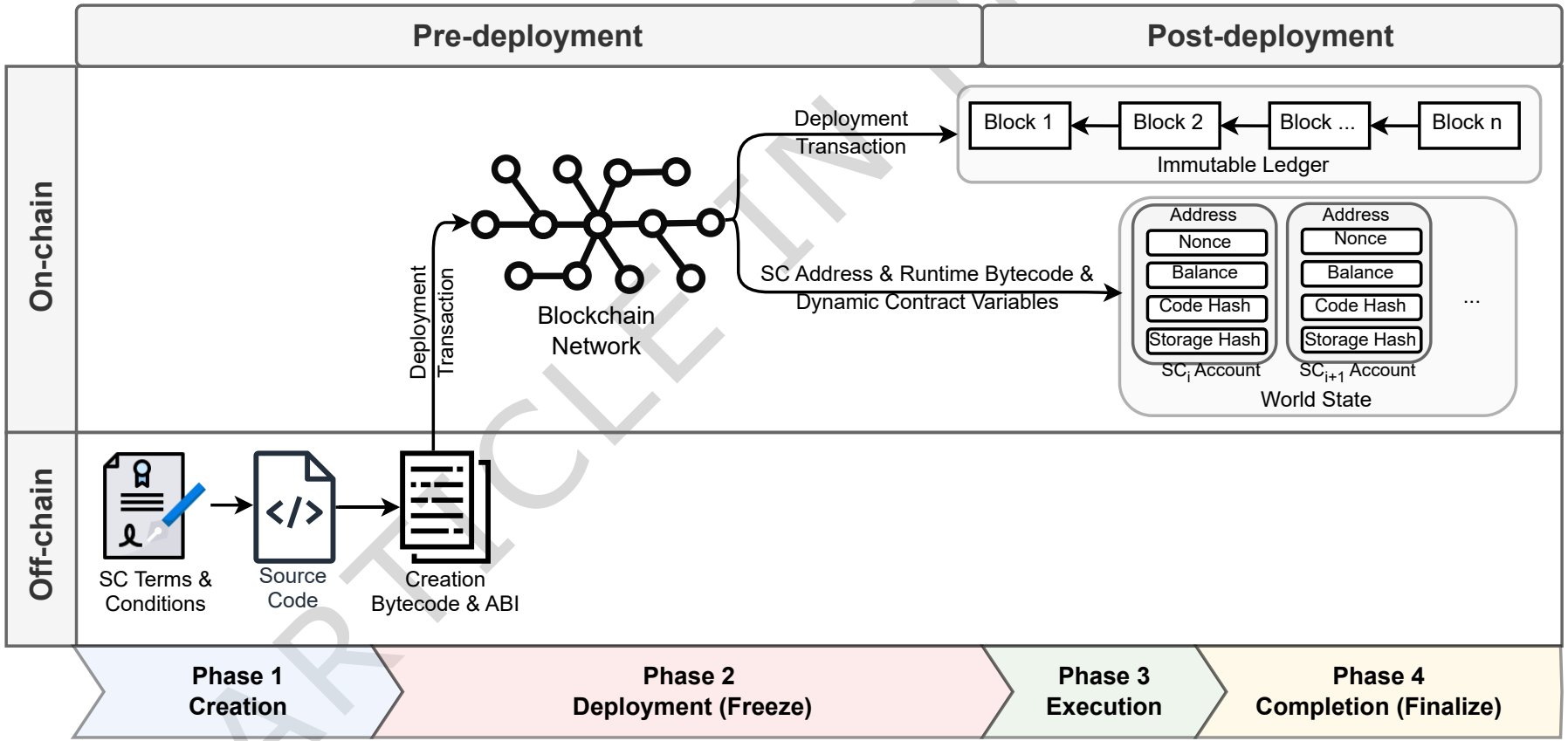
The authors would like to acknowledge the support of the King Fahd University of Petroleum and Minerals (KFUPM), Saudi Arabia, in the development of this work.

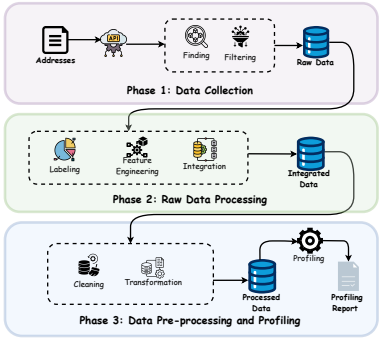
Author Contributions

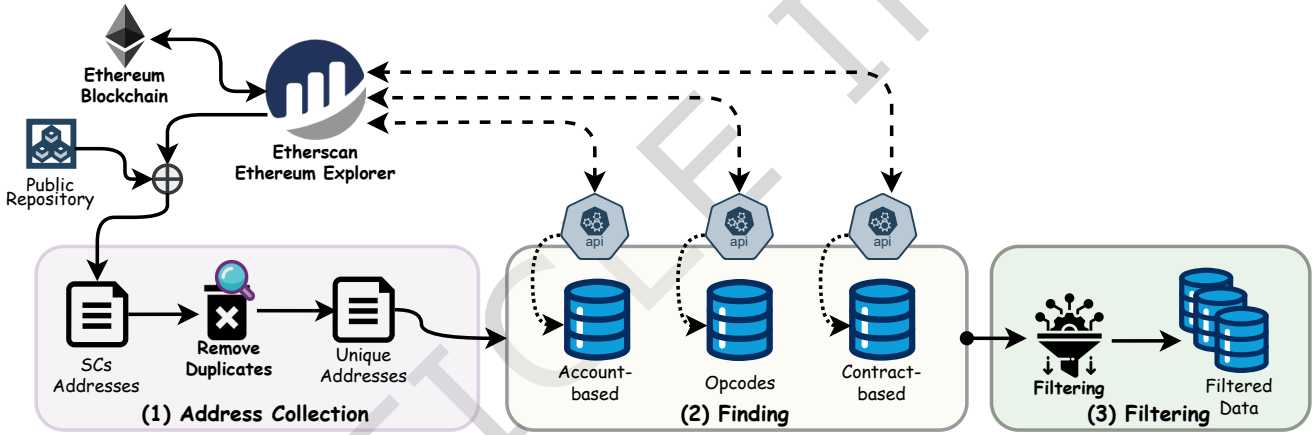
Conceptualization: S.J.A., H.A., M.H., Data curation: S.J.A., Formal analysis: S.J.A., Investigation: S.J.A., Methodology: S.J.A., H.A., Resources: S.J.A., Software: S.J.A., Supervision: H.A., M.H., Validation: S.J.A., H.A., M.H., Visualization: S.J.A., Writing—original draft: S.J.A., Writing—review & editing: S.J.A., H.A., M.H.

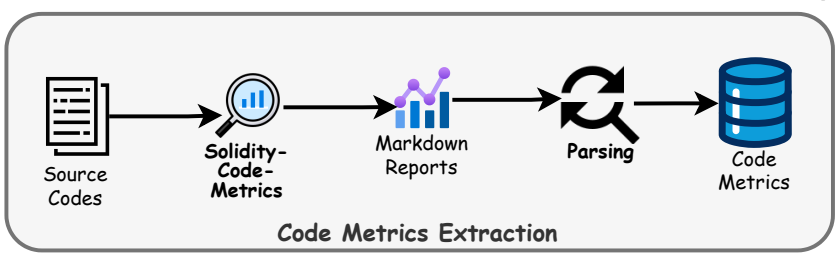
Competing Interests

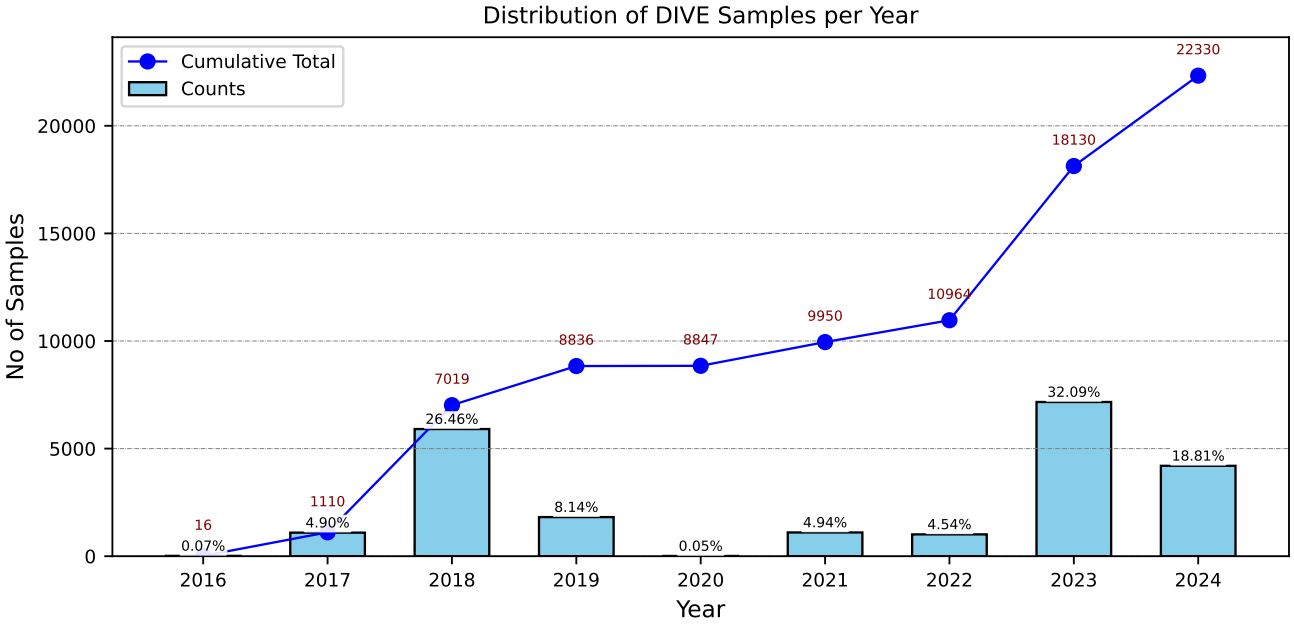
The authors declare no competing interests.



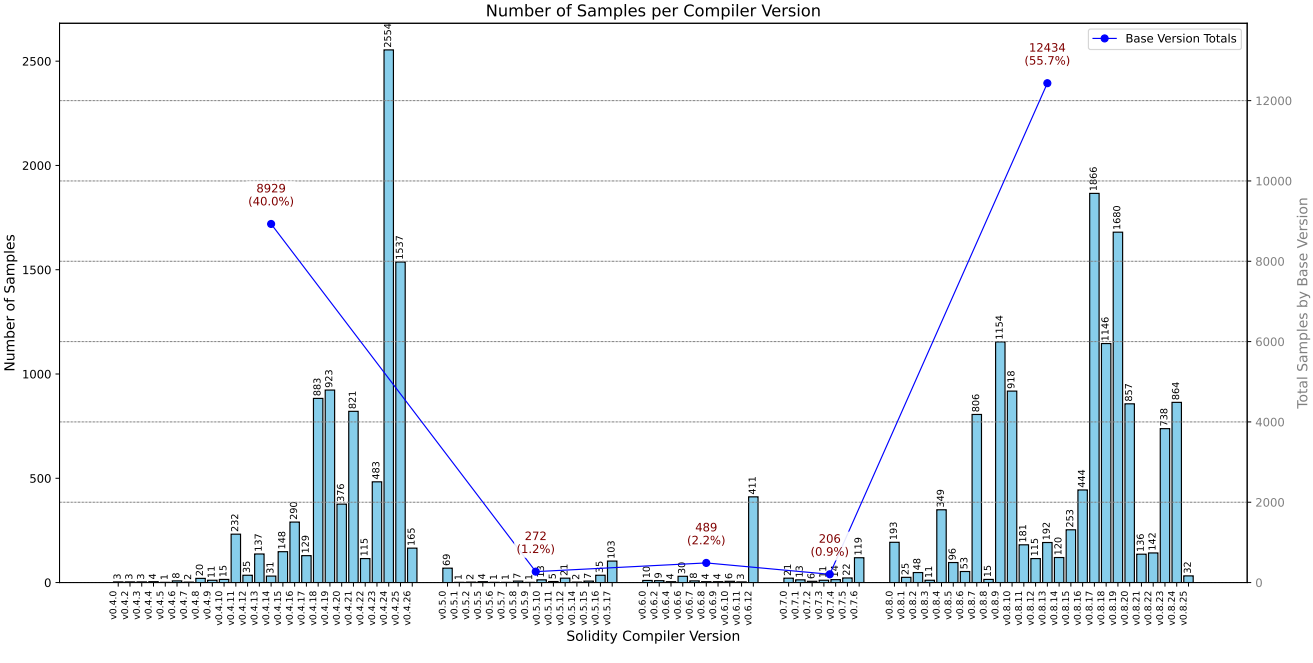




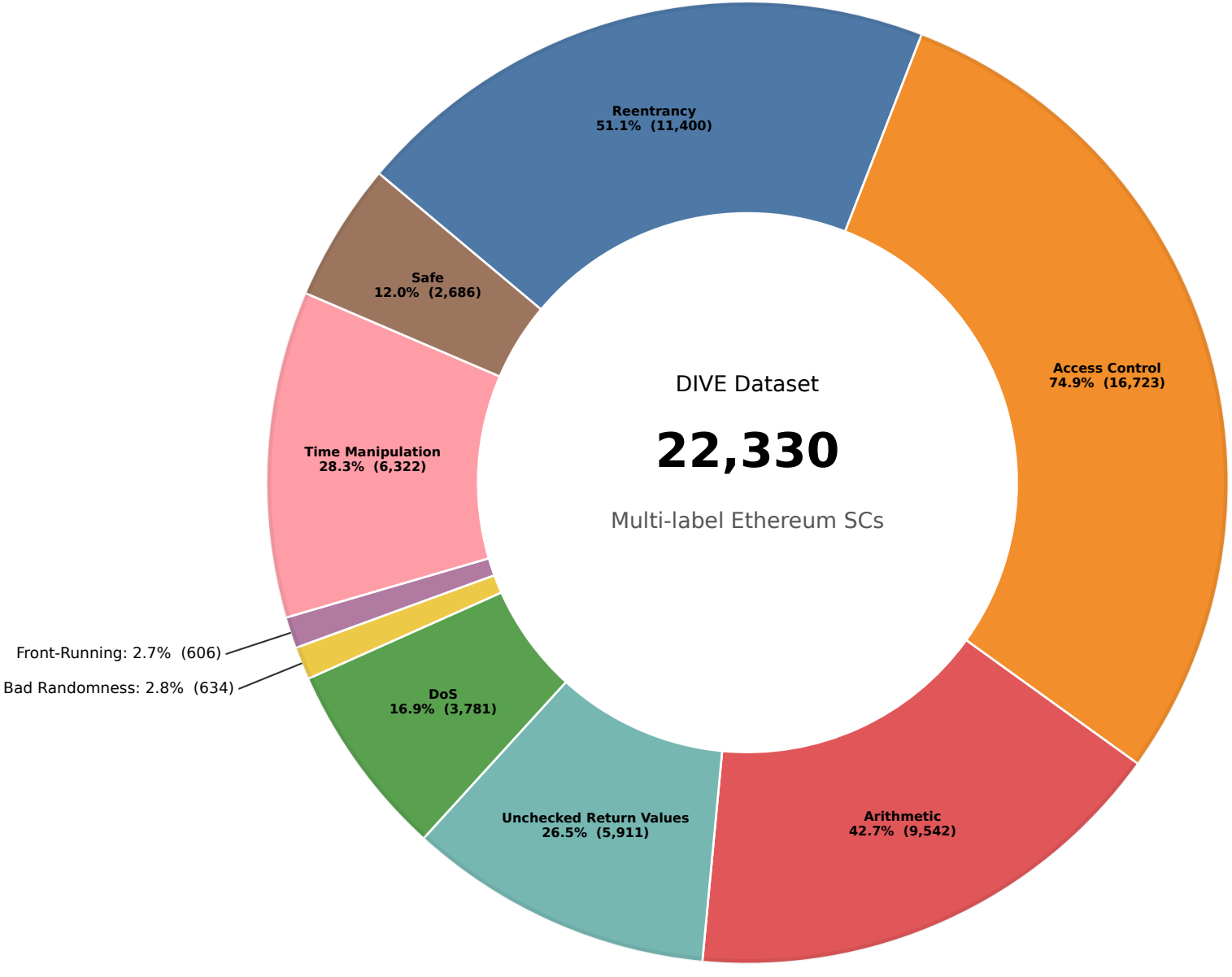


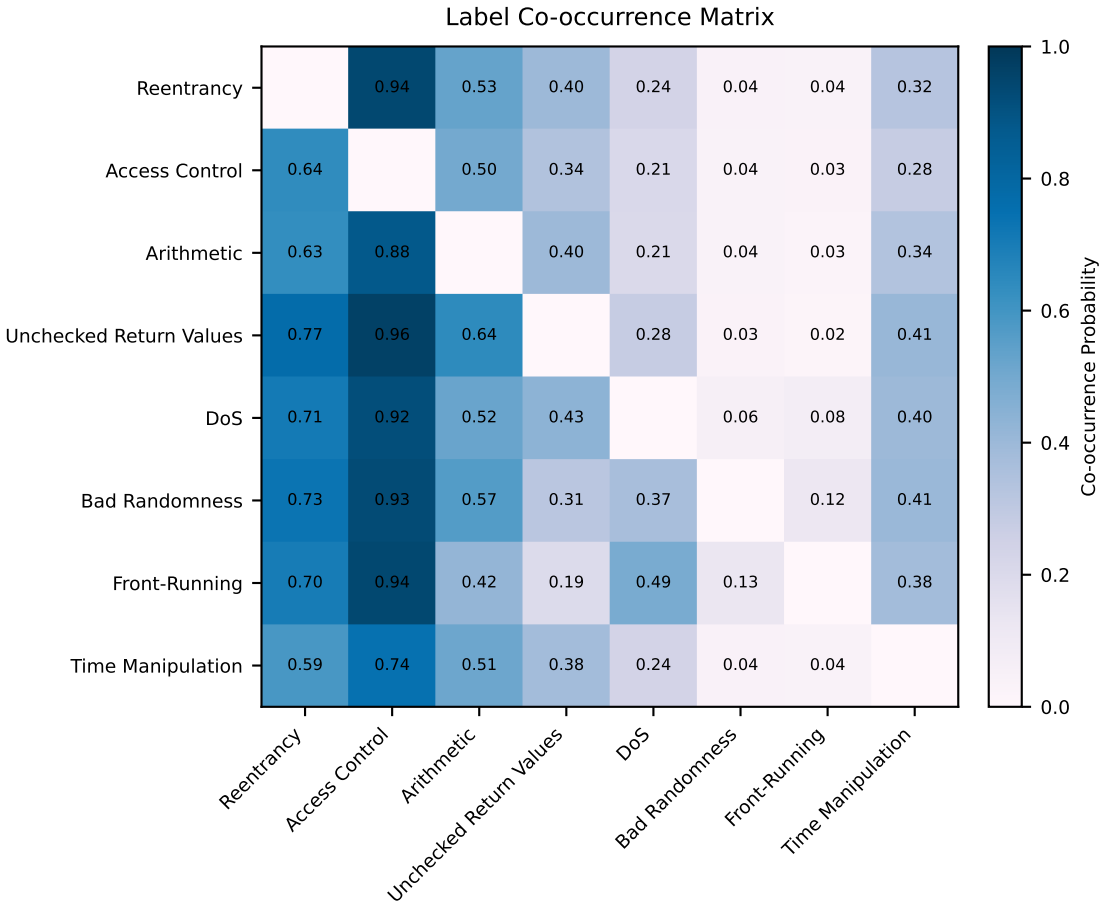


(a) Distribution of DIVE samples per year.

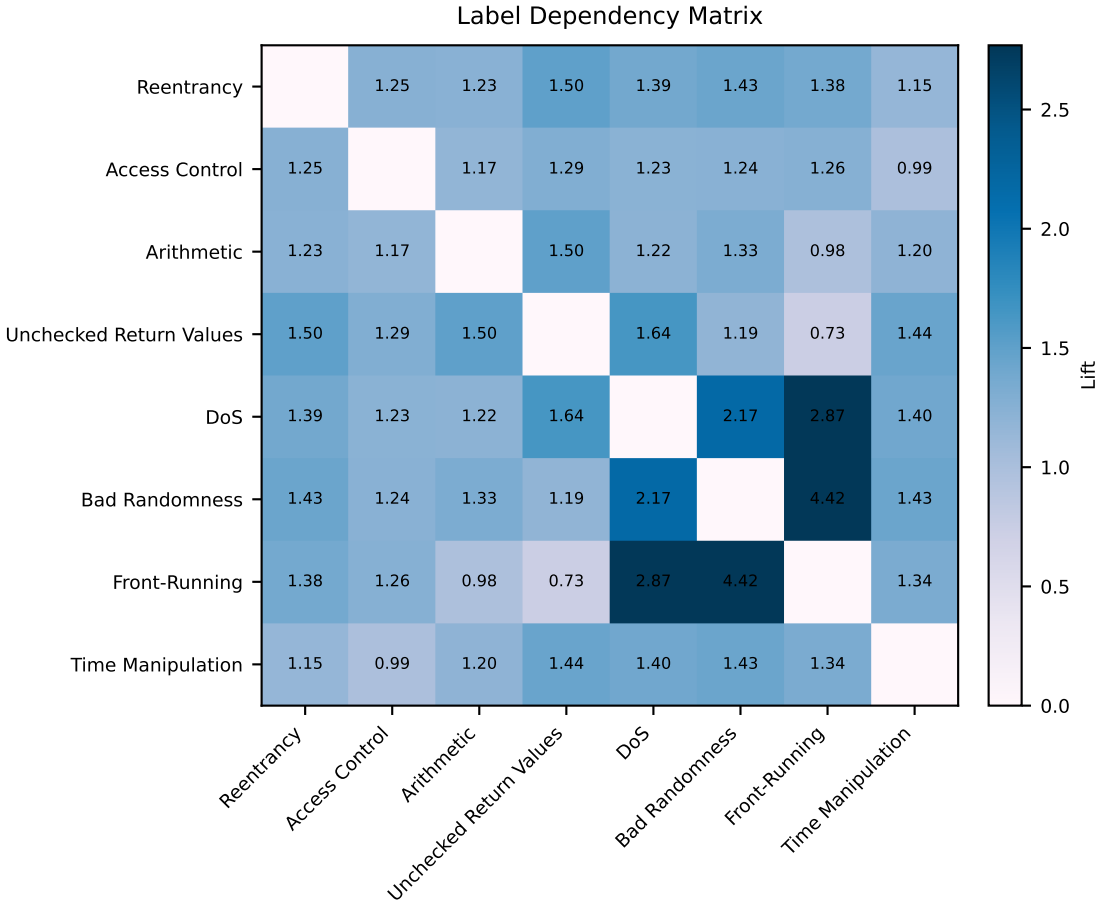


(b) Number of samples per Solidity compiler version.


Reentrancy
51.1% (11,400)
Access Control
74.9% (16,723)
Safe
12.0% (2,686)
Time Manipulation
28.3% (6,322)
DIVE Dataset
22,330
Multi-label Ethereum SCs
Front-Running: 2.7% (606)
Bad Randomness: 2.8% (634)
DoS
16.9% (3,781)
Unchecked Return Values
26.5% (5,911)
Arithmetic
42.7% (9,542)



(a) Conditional co-occurrence of vulnerability labels.



(b) Label dependency measured using lift.