FORGE: An LLM-driven Framework for Large-Scale Smart Contract Vulnerability Dataset Construction

Jiachi Chen[1,2], Yiming Shen[1], Jiashuo Zhang[3*], Zihao Li[4], John Grundy[5], Zhenzhe Shao[1]
Yanlin Wang[1], Jiashui Wang[6], Ting Chen[7,8], Zibin Zheng[1]
[1]*Sun Yat-sen University, Zhuhai, China*
[2]*The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China*
[3]*School of Computer Science, Peking University, Beijing, China*; *Corresponding author
[4]*The Hong Kong Polytechnic University, Hong Kong, China*
[5]*Monash University, Melbourne, Australia*; [6]*Zhejiang University, Hangzhou, China*
[7]*University of Electronic Science and Technology of China, Chengdu, China*
[8]*Kashi Institute of Electronics and Information Industry, Kashi, China*
chenjch86@mail.sysu.edu.cn,shenym7@mail2.sysu.edu.cn,zhangjiashuo@pku.edu.cn
zi-hao.li@connect.polyu.hk,john.grundy@monash.edu,shaozhzh3@mail2.sysu.edu.cn
yanlin-wang@outlook.com,12221251@zju.edu.cn,brokendragon@uestc.edu.cn,zhzibin@mail.sysu.edu.cn

ABSTRACT

High-quality smart contract vulnerability datasets are critical for evaluating security tools and advancing smart contract security research. Two major limitations of current manual dataset construction are (1) labor-intensive and error-prone annotation processes limit the scale, quality, and evolution of the dataset, and (2) absence of standardized classification rules results in inconsistent vulnerability categories and labeling results across different datasets. To address these limitations, we present FORGE, the first automated approach for constructing smart contract vulnerability datasets. FORGE leverages an LLM-driven pipeline to extract high-quality vulnerabilities from real-world audit reports and classify them according to the Common Weakness Enumeration (CWE), the most widely recognized classification in software security. FORGE employs a divide-and-conquer strategy to extract structured and self-contained vulnerability information from unstructured audit reports. Additionally, it uses a tree-of-thoughts technique to classify the vulnerability information into the hierarchical CWE classification. To evaluate FORGE's effectiveness, we run FORGE on 6,454 real-world audit reports and generate a dataset comprising 81,390 solidity files and 27,497 vulnerability findings across 296 CWE categories. Manual assessment of the dataset demonstrates high extraction precision and classification consistency with human experts (precision of 95.6% and inter-rater agreement k-α of 0.87). We further validate the practicality of our dataset by benchmarking 13 existing security tools on our dataset. The results reveal the significant limitations in current detection capabilities. Furthermore, by analyzing the severity-frequency distribution patterns through a unified CWE perspective in our dataset, we highlight inconsistency between the current smart contract research focus and the priorities identified from real-world vulnerabilities. This analysis also reveals key differences from traditional software security concerns, providing practical implications for improving the smart contract security ecosystem.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*.

KEYWORDS

Smart contracts, Blockchain, Vulnerability dataset, LLM

ACM Reference Format:
Jiachi Chen[1,2], Yiming Shen[1], Jiashuo Zhang[3*], Zihao Li[4], John Grundy[5], Zhenzhe Shao[1], Yanlin Wang[1], Jiashui Wang[6], Ting Chen[7,8], Zibin Zheng[1] . 2025. FORGE: An LLM-driven Framework for Large-Scale Smart Contract Vulnerability Dataset Construction. In *Proceedings of IEEE/ACM 48th International Conference on Software Engineering (ICSE '26).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Smart contracts on blockchain platforms have revolutionized digital asset management and decentralized applications [6, 88], with the total value locked (TVL) exceeding $117 billion [15]. However, this rapid growth has been accompanied by significant security challenges. In 2024, there were more than 760 on-chain security incidents, resulting in $2.36 billion in losses [8]. These security incidents caused by evolving attack vectors [90] underscore the critical importance of robust vulnerability detection mechanisms in the smart contract ecosystem [10, 78].

High-quality vulnerability datasets are essential for developing security tools and advancing security research for smart contracts [64, 85, 87]. However, current approaches to smart contract vulnerability dataset construction face two critical limitations. ***First, the manual process of dataset construction is labor-intensive and error-prone***, which limits the scale, quality, and ability to keep up with the rapidly changing vulnerability landscape. For instance,

ICSE '26, April 12-18, 2026, Rio de Janeiro, Brazil

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

a recent smart contract vulnerability dataset construction [87] required 44 person-months of manual effort to compile 1,618 vulnerabilities across 682 DApps, introducing potential quality concerns and making it difficult to evolve when new vulnerabilities emerge. ***Second, the absence of standardized classification rules*** leads to inconsistent vulnerability categories and labeling results across different datasets. Existing datasets employ different vulnerability classifications, such as Smart Contract Weakness Classification (SWC) [69] or DASP10 [22], which may overlap or describe the same vulnerability but have different names [67]. For instance, a vulnerability labeled as "Front-running" [22] might appear as "Transaction Order Dependence (TOD)" or "Block Manipulation" [?] in another, despite describing the same underlying issue at different granularities. The lack of unified vulnerability classification rules complicates the development and evaluation of vulnerability detection tools and leads to inconsistent vulnerability management practices [36].

To address these key limitations, we present FORGE, the first automated framework for constructing comprehensive and high-quality smart contract vulnerability datasets. FORGE employs an LLM-driven pipeline to automatically extract vulnerability information from real-world audit reports [83], significantly reducing the manual effort required for dataset construction. To ensure standardized classification, we integrate the Common Weakness Enumeration (CWE) [38] - the most widely recognized vulnerability classification in software security [33, 57]. Specifically, to effectively extract vulnerability information from real-world audit reports, FORGE implements a divide-and-conquer strategy through a map-reduce paradigm to effectively process lengthy audit reports and extract structured self-contained vulnerability information. To accurately classify the extracted vulnerabilities into the CWE classification, FORGE introduces a tree-of-thoughts [81] reasoning technique that leverages in-context learning to classify vulnerabilities hierarchically to appropriate CWE categories. Finally, FORGE collects the source codes related to the extracted vulnerabilities and constructs CWE-labeled vulnerability entries within the vulnerability dataset.

To construct a comprehensive dataset and evaluate FORGE's effectiveness, we first collect and filter out 6,454 smart contract audit reports from 47 security teams [19]. Using the reports as input, FORGE took only 229.5 hours to construct a large-scale dataset comprising 27,497 vulnerability findings within 81,390 Solidity files from real-world projects and covering 296 CWE categories. These files averaged 2,575 lines of code, with 59.0% using the latest solidity compiler version (v0.8+). This efficient and automated process represents a significant improvement over existing manual dataset construction practices like DAppSCAN, which requires 44 person-months effort to build a dataset containing 39,904 files with 1,618 vulnerabilities from 25 categories [87]. We evaluate the performance of FORGE in extracting vulnerability-related information entities, achieving a Macro-F1 score of 86.1%, with an average precision of 95.6%. Additionally, we assess the consistency of CWE classification between FORGE and human experts, obtaining a high Krippendorff's α coefficient of 0.87 [28]. Moreover, to validate the practicality of our dataset, we benchmark 13 widely used security tools. The results indicate their limited effectiveness, with the highest F1 score reaching 18.59% and an average of 5.06% across all tools.

To further provide insights with our dataset for security practitioners, we conduct an analysis of the smart contract vulnerability landscape, including visualizations of risk prioritization and comparative studies. By analyzing the severity-frequency distribution through a unified CWE classification, we reveal not only inconsistency between previous smart contract research focus and real-world sourced vulnerability priorities but also distinct characteristics compared to traditional software security concerns.

The key contributions of this research include:

- We introduce FORGE, the first LLM-based framework designed to automatically construct smart contract vulnerability datasets by extracting vulnerability information from audit reports and classifying it into the CWE classification.
- We used FORGE to construct a dataset that includes 27,497 CWE-annotated vulnerability findings from 81,390 real-world Solidity files. This dataset features an average of 2,575 lines of code per project, with 59.0% of projects using the latest Solidity compiler versions (v0.8+).
- We highlight the limitations of existing security tools and conduct an empirical analysis that combines the frequency and severity of each vulnerability from a CWE perspective to derive insights from previous research on both smart contracts and traditional software.
- We have made our dataset, experimental results, and the source code of FORGE available at https://github.com/shenyimings/FORGE-Artifacts.

2 BACKGROUND AND MOTIVATION

2.1 Common Weakness Enumeration (CWE)

Common Weakness Enumeration (CWE) is a dictionary of software and hardware vulnerabilities, regularly maintained by the community to reflect emerging security issues [38, 67].

CWE-1000: Research Concepts [40] view provides a comprehensive classification system that follows a deep tree structure with four abstraction levels that organize weaknesses from abstract concepts to specific implementations: *Pillar*, *Class*, *Base*, and *Variant*. At the highest level, there are ten *Pillar* level vulnerability categories, representing fundamental vulnerabilities that cannot be further abstracted. For example, *CWE-284: Improper Access Control* [43] is one of the ten *Pillar* level weaknesses, with *CWE-287: Improper Authentication* [44] as its *Class*-level child. Moving down the hierarchy, the *Base* level introduces specific behaviors and properties as illustrated by *CWE-295: Improper Certificate Validation* [45]. This specificity is further refined at the *Variant* level, where technology-specific details and implementations are addressed, as demonstrated by *CWE-298: Improper Validation of Certificate Expiration* [46].

The *CWE Classification* process classifies diverse real-world security issues to standardized CWE entries. This process identifies and codifies the underlying flaws or errors responsible for exploitable security risks [39]. The hierarchical structure of CWE allows for the classification of nearly any software vulnerability to an appropriate entry. Prior studies [61, 67, 89] and audit reports *e.g.* [9] have adopted CWE as an identifier for smart contract vulnerabilities.

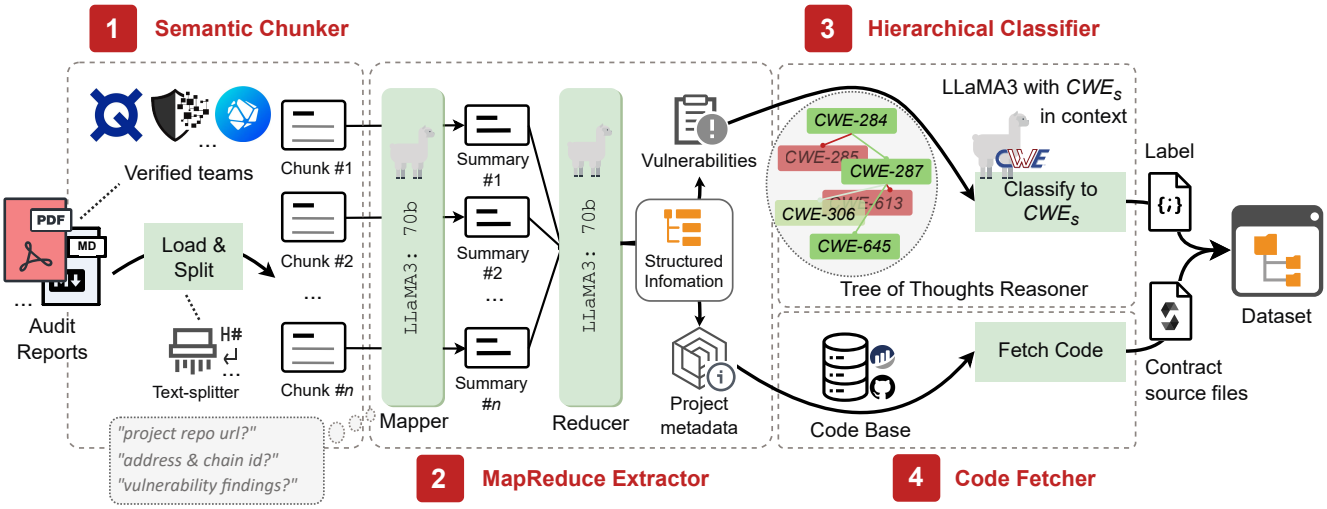

Figure 1: Overview of FORGE framework.

2.2 Smart Contract Audit Reports

Smart contract auditing is a rigorous security assessment process conducted by professional audit teams (*e.g.*, Etherscan-verified audit teams [19]) to identify security issues and document vulnerabilities in detailed audit reports [24]. These expert-validated reports not only detail vulnerability attack vectors but also provide essential context about their discovery and potential impact, making them ideal sources for systematic vulnerability analysis [25, 84]. They aim to identify CWE-related vulnerabilities in smart contracts and may recommend how to fix them.

2.3 LLMs for Dataset Construction

Large Language Models (LLMs) are advanced machine learning models trained on extensive datasets of text and code, capable of understanding and generating human-like language across diverse domains [27]. Notable examples include *GPT-4* [55], *Claude3.5* [4], and *Llama3* [2]. These powerful models have demonstrated remarkable capabilities in natural language processing [71, 79], program understanding [86], and security analysis [33, 68].

Existing manual dataset construction processes are highly labor-intensive and error-prone, which largely limits the scale, quality, and evolution of the dataset. The promising capabilities of LLMs in program comprehension and information extraction [32] demonstrate the potential to reduce the manual effort required by the current dataset construction processes. This motivates us to investigate whether an LLM-driven pipeline can be leveraged to achieve automated vulnerability dataset construction.

Applying LLMs to build CWE-labeled smart contract vulnerability datasets from complex real-world audit reports introduces three key challenges: *(1) Complexity of audit reports.* For instance, the Trail of Bits' audit report for Uniswap v4 Core [60] spans 63 pages, documenting project metadata and vulnerability findings alongside extensive audit disclaimers, testing methodology descriptions, and code architecture explanations. Such complexity necessitates a systematic approach to extract and aggregate vulnerability information efficiently. *(2) Complexity of the CWE hierarchy.* With over 900 weakness types across multiple abstraction levels and dimensions [40]. Traditional approaches such as keyword match [37] and knowledge graph [23] often lead to inconsistent or superficial results[33, 65]. *(3) Limitations of LLMs.* LLMs exhibit constraints in handling domain-specific technical tasks. Their effectiveness is hampered by the limited context window and the "lost in the middle" phenomenon [31], which impairs the processing of lengthy audit reports and leads to overlooked content details [26]. Moreover, their inherent tendency to generate hallucinated content [86] could compromise the precision of vulnerability classification.

3 FORGE FRAMEWORK

To automate smart contract vulnerability dataset construction and to improve scaling, quality, and evolution, we present FORGE, an end-to-end framework that leverages LLMs to build CWE-labeled datasets from real-world audit reports.

3.1 Overview

The workflow of FORGE is outlined in Figure 1. The process begins with the *Semantic Chunker*, which takes an audit report as input and segments it into self-contained chunks based on semantic boundaries in the report file. The *MapReduce Extractor* then processes individual chunks to extract vulnerability information (map phase) and aggregate the results into structured summaries (reduce phase). The *Hierarchical Classifier* employs the tree-of-thoughts [81] approach to systematically classify vulnerabilities into the CWE_s (software-related entries of CWE) hierarchy, utilizing in-context learning [17] with domain knowledge injection at each decision point. Finally, the *Code Fetcher* module retrieves and integrates relevant smart contract source files, complementing the extracted vulnerability information to produce structured and classified vulnerability findings for vulnerability dataset construction.

3.2 Phase 1: Semantic Chunker

To address the token length limitations of LLMs, while preserving semantic integrity, we first apply a *Semantic Chunker* module. This splits long-form audit reports into self-contained, semantically

coherent chunks by segmentation and chunk size optimization. Specifically, a chunk is a text segment that (1) preserves complete semantic information related to a vulnerability and (2) length falls within the LLM's processing threshold.

To split the audit report into self-contained chunks, FORGE first converts the input report file into a unified markdown format using `PyMuPDF` [59], preserving the file structure of headings and content. Then, it operates in the following steps: *First*, the document's inherent structure is leveraged to segment content along natural semantic boundaries. These boundaries include document headings, paragraph breaks, and Unicode-defined text units (from grapheme clusters to sentences) to preserve the semantic hierarchy of the source document. *Then*, a size verification ensures all chunks remain within the LLM's token limit. If necessary, oversized chunks are further divided at appropriate boundaries recursively.

3.3 Phase 2: MapReduce Extractor

To systematically process diverse and lengthy audit reports, FORGE employs the MapReduce Extractor following the semantic chunking phase. This module adopts a divide-and-conquer strategy, which first extracts structured project metadata and vulnerability information from these chunks (*i.e.*, the map stage) and then merges the vulnerability information among chunks (*i.e.*, the reduce stage). Specifically, the project metadata encompasses blockchain network, on-chain address, GitHub URL, and commit ID, which is necessary for accessing the source code associated with identified vulnerabilities. The vulnerability information includes the title, description, severity, and location, elucidating the vulnerability's attack vector, prerequisites, and potential impacts of exploitation. In the map stage, FORGE extracts information for each chunk c_i:

$$s_i = f_{\text{map}}(c_i, \mathcal{Q}; \theta) \tag{1}$$

where $\mathcal{Q}$ denotes the information types that the LLM needs to extract, including project metadata and vulnerability findings. The LLM-powered mapping function f_{map}, parameterized by θ, performs extraction of input chunks c_i, producing outputs $\{s_1, \ldots, s_N\}$ conforming by $\mathcal{Q}$.

In the reduce stage, the final structured information is generated through:

$$a = f_{\text{reduce}}(\{s_1, \cdots, s_K\}, \mathcal{P}, \mathcal{V}; \theta) \tag{2}$$

where $\mathcal{P}$ denotes project metadata comprising source code identifiers (including GitHub repository URLs with specific commit hashes or on-chain contract addresses), and $\mathcal{V}$ represents vulnerability attributes containing title, description, severity level, and affected code locations. The f_{reduce} operation first concatenates map results $\{s_1, \cdots, s_K\}$ until reaching the `chunk_length` threshold defined in Section 3.2, then instruct LLM parameterized by θ to deduplicate and merge the map results to a structured JSON output a. Figure 2 illustrates this structured output schema.

In cases where the combined length of map summaries (Σs_i) exceeds the model's context window limit `chunk_length`, the MapReduce extractor employs a two-stage reduction strategy. First, it partitions the N summaries into K groups, where each group's total length remains within `chunk_length`. The system then performs K independent reduce operations, each producing a partially

```
project_info:
    url: string,
    commit_id: string,
    address: string,
    chain: string
findings:
    id: int,
    title: string,
    description: string,
    severity: enum(critical, high, medium, low,
        info),
    location: string
```

Figure 2: Structured information

structured result containing project metadata and vulnerability information.

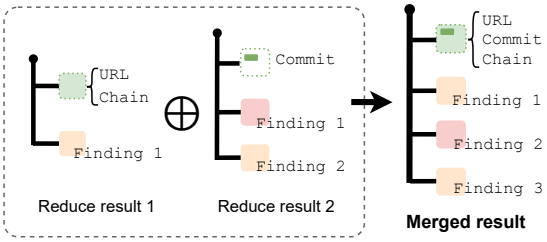

Figure 3: A diagram of JSON merge operation

To merge these K partial results into a final output, we define an operator $\oplus$ that combines both project metadata ($\mathcal{P}$) and vulnerability information ($\mathcal{V}$) from two reduced JSON results in a_i, a_j:

$$a_i \oplus a_j = \{\mathcal{P}_i \cup \mathcal{P}_j, \mathcal{V}_i \cup \mathcal{V}_j\} \tag{3}$$

As illustrated in Figure 3, the merge operation discards empty fields and combines non-conflicting fields. In cases of conflicts, the system adopts information from chunks that appear *earlier* in the original document sequence. This choice is based on experimental observations indicating that project metadata, *e.g.* URLs, commit IDs, and addresses, typically appears in the earlier sections of audit reports. Consequently, the information present in the earlier chunks is considered more reliable. It has a lower likelihood of hallucination by the LLM compared to the information found in later chunks.

Our map-reduce-based design ensures the completeness and consistency of extracted information while handling long-form reports that exceed the model's context limitations.

3.4 Phase 3: Hierarchical Classifier

After completing smart contract vulnerability information extraction, we then classify the structured vulnerability findings into CWE categories. Given the extensive parent-child structure of CWE classifications as detailed in Section 2.3, our *Hierarchical Classifier* module integrates an LLM-driven tree-of-thoughts (ToT) [81] reasoning process to navigate the complex hierarchical structure of CWE, and utilizes In-Context Learning (ICL) [17] for CWE knowledge injection for accurate classification.

3.4.1 Pre-processing. To better characterize software-level vulnerabilities relevant to smart contracts, we refine the *CWE-1000: Research Concepts* view [40], which contains both hardware and software vulnerability categories. Specifically, we identified and filtered out 108 hardware-related entries (*e.g.*, *CWE-1192: Improper Identifier for IP Block used in System-On-Chip* [41]) that are irrelevant to software security assessment. We denote this refined subset as CWE_s, which forms the foundation for the *Hierarchical Classifier*. Unless otherwise specified, all subsequent references to CWE categories in this paper refer to entries within this software-focused subset.

Additionally, following guidance from the CWE community [39], we labeled each entry in CWE_s with mapping notes indicating whether the entry is suitable as a final category for smart contract vulnerability. These labels guide our classifier by signaling which CWE_s nodes can serve as valid endpoints during the classification process, enabling proper fallback to higher-level categories when more specific classifications are not appropriate. The detail of this subset with mapping notes is provided in our online repository.

3.4.2 Tree-of-Thoughts (ToT) Reasoning. Our classification approach leverages the inherent tree structure of CWE, where vulnerabilities are organized in a hierarchical taxonomy from pillar-level categories to concrete leaf nodes [40]. By adopting a tree-of-thoughts (ToT) reasoning framework [81], FORGE can systematically explore classification decisions from general to specific CWE categories while maintaining previous decision paths.

Our method transforms the vulnerability classification challenge into a guided tree node search problem, where the LLM – with CWE_s knowledge in context – serves as a pathfinder that traverses through the CWE_s hierarchy. At each level, LLM considers the vulnerability information alongside the local structure of candidate CWE_s categories, determining whether to advance to more specific categories or fall back to previous nodes when appropriate.

Algorithm 1 Tree-of-Thoughts Reasoner

Require: vuln_info; llm; CWE_s; k, l
1: **global** path ← []
2: **function** CLASSIFY(node, l)
3: *children* ← GETCHILDREN(CWE_s, node)
4: **if** children is empty **then return**
5: **end if**
6: *fallback_node* ← []
7: **if** node is MappingAllowd **then** fallback_node ← node
8: **end if**
9: *prompt* ← CONSTRUCTPROMPT(vuln_info, fallback_node, children, k)
10: selected_node(s) ← PARSE(llm(*prompt*))
11: **if** fallback_node in selected_node(s) **then return**
12: **end if**
13: path.append([(l, selected_node(s))])
14: CLASSIFY(selected_node(s), l+1)
15: **end function**
16: **initialize** vuln_info, llm, CWE_s, k
17: CLASSIFY(root of CWE_s, 0)
18: **return** path

We formalize the classification process as a sequence of structured reasoning steps, as presented in Algorithm 1. It takes the vulnerability information *vuln*, the language model *llm*, the pruned CWE hierarchy CWE_s, the number of most relevant child nodes to be selected at each level k and the current level in the hierarchy l as inputs. The output is the classification *path*.

The algorithm performs a recursive traversal of the CWE_s hierarchy, starting from the root node and progressively classifying the vulnerability into more specific categories. At each level l, *llm* functions as a classifier, taking prompt as input: (1) the vulnerability title and description, (2) the current set of candidate child nodes with their descriptions, and (3) a parameter k specifying the number of relevant nodes to select. *llm* leverages its semantic understanding capabilities to reason between the vulnerability information and each candidate node's characteristics, outputting the k most relevant child CWE_s nodes.

3.4.3 Fallback Strategy. In some cases, the most appropriate CWE_s category for a vulnerability may not be a leaf node but a more abstract node closer to the root. To handle this, we introduce a fallback strategy. For CWE_s nodes that are labeled as mapping allowed, we add them to the $fallback\_node$ list in addition to their child nodes (lines 8-10 in Algorithm 1). If the *llm* selects a fallback node, the recursion terminates, and the final classification path is returned (lines 13-15). Otherwise, the recursion continues to level $l + 1$ if child nodes are selected by *llm* (lines 16-17).

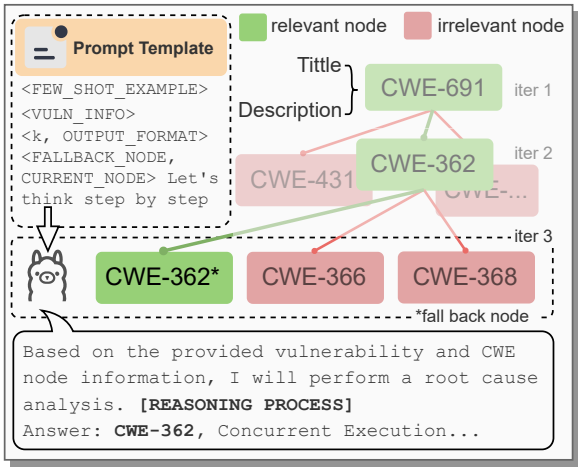

Figure 4: An example of ToT workflow

Figure 4 shows an example of the CWE_s classification workflow. For a vulnerability finding related to a marketplace contract affected by potential front-running manipulations due to the absence of a minimum swap output value enforcement mechanism, the classification proceeds as follows. With $k = 1$, in the first iteration, it is classified into *CWE-691: Insufficient Control Flow Management* [49] out of the 10 pillar categories. In the second iteration, from the child nodes of *CWE-691*, the vulnerability is further classified into *CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')* [47]. *CWE-362* is labeled as mapping allowed, indicating that it is a suitable category for smart contract vulnerability mapping. Therefore, in the third iteration,

CWE-362 is added to the selection list as a fallback node along with its child nodes. LLM selects CWE-362 again, and the iteration ends. The final classification path is *CWE-691→CWE-362*.

3.5 Phase 4: Code Fetcher

The metadata of target projects obtained from the extractor is then processed by the *Code Fetcher*. This retrieves the corresponding smart contract source code from trusted public repositories – GitHub and blockchain explorers such as Etherscan [20] and Bscscan [5]). The process of dealing with on-chain code is straightforward due to blockchain's immutable nature. Source code at a specific contract address directly corresponds to the version affected by the reported vulnerability. For smart contract projects hosted on Github repositories, developers typically create an *audit* branch for review [77], which undergoes a thorough examination by security auditors. Subsequently, developers implement fixes through new commits. To maintain precise vulnerability information tracking, our *MapReduce* extractor module captures the `commit-id` (typically in the form of a hash or a descriptive name) of the audited codebase, allowing the *Code Fetcher* module to retrieve the exact version of the source code files. Finally, the *CodeFetcher* combines the collected source code files and the extracted vulnerability information into vulnerability entities within the constructed dataset.

4 EVALUATION

In this section, we intend to answer the following key research questions (RQs):

- **RQ1. Can FORGE effectively generate a large-scale smart contract vulnerability dataset from real-world audit reports?** We run FORGE on 6,454 audit reports to construct a dataset, analyzing the statistics and comparing them with existing datasets.
- **RQ2. How does FORGE perform in vulnerability information extraction?** We assess the effectiveness of FORGE framework in information extraction.
- **RQ3. How consistently does FORGE perform in vulnerability classification against human experts?** We assess the effectiveness of FORGE framework for vulnerability classification.
- **RQ4. How practical is the FORGE dataset for evaluating current smart contract vulnerability detection tools?** We leverage our large-scale dataset to assess the efficacy and limitations of existing security tools, highlighting the practicability of the dataset.

Data Collection. To curate our experimental dataset, we collect 10,312 publicly available audit reports from 47 renowned auditing teams verified through Etherscan [19]. Since our dataset construction requires both vulnerability information and the corresponding source code, we filter out audit reports where the source code is not publicly accessible. We apply a regular expression matching script to identify valid on-chain addresses and source code repository URLs, ultimately obtaining 6,454 documents for further analysis.

Experiment Setup. We conducted our experiments on a CentOS 7.9 server equipped with 128 Intel(R) Xeon(R) Platinum 8376H CPUs @ 2.60GHz, 512GB RAM, and 2 NVIDIA A800 80GB PCI GPUs. We employ *Llama3:70b-instruct-q8_0* as the foundation model for the *MapReduce Extractor* and *Hierarchical Classifier* modules, with a chunk length of 4,096 tokens, the number of a most relevant child node k set to 1 and a default temperature setting of 0.8 [3].

```
{"path": "reports/xrpn.pdf",
 "project_info":
   {"url": "n/a",
    "commit_id": "n/a",
    "address": "0xC07c894D01A8785CB289620C...",
    "chain": "bsc",
    "compiler_version": ["v0.8.6+commit..."],
    "project_path": {"XRPNATION": "..."}},
 "findings":
   [{"id": 1,
     "category":
       {"1": ["CWE-284"],"2": ["CWE-269"],...},
     "title": "Contract owner can exceed...",
     "description": "Owner has authority...",
     "severity": "critical",
     "location": "contract.sol#L642"}]}
```

Figure 5: An example record of FORGE dataset

4.1 RQ1: Effectiveness of FORGE

We begin by running FORGE on our previously collected 6,454 audit reports. This took a total of 229.5 hours, with an average processing time of 127.8 seconds per report. This includes 45.0 seconds for completing a vulnerability information extraction task and 18.3 seconds for each vulnerability classification.

Table 1: Overview of our FORGE dataset

Statistics	Numbers
Total audit reports	6,454
Total DApp projects	6,579
Total solidity files	81,390
Average solidity files in a project	12
Average line of code in a project	2,575
Compiler Version 0.4+	270
Compiler Version 0.5+	478
Compiler Version 0.6+	1,524
Compiler Version 0.7+	360
Compiler Version 0.8+	3,791
Other Compiler Version	31
Total vulnerability findings	27,497

The dataset built by FORGE encompasses 81,390 Solidity files, covering 27,497 vulnerabilities with CWE labels. Essential information for each project is stored in 6,454 JSON files. Table 1 outlines the key parameters of our dataset, highlighting that each project contains an average of n Solidity smart contract files, with a mean of 2,575 lines of code per project. Notably, the most prevalent compiler version in our dataset is 0.8+ (59.0%). Compared to the widely-used SmartBugs dataset [18], where contracts average only 204 lines of code and over 90% use outdated compiler versions (0.4+) [87], **our real-world sourced dataset exhibits significantly higher complexity and comprehensiveness**. Similarly, it also shows **a substantial increase in scale and coverage** compared to human-annotated datasets *e.g.* DAppSCAN, which contains 39,904 files with 1,618 vulnerabilities, with 35.2% of them using compiler 0.8+ [87].

Figure 5 showcases an example of a typical record in our dataset. Each entry is comprised of several key fields that provide comprehensive information about the audited smart contract and its vulnerabilities. The "path" field identifies the corresponding reports, while the "project_info" field captures essential metadata about the DApp, including its URL, commit ID, contract address, blockchain network, compiler version, and Solidity file paths. The "findings" field is an array where each element represents a distinct vulnerability. These vulnerability entries are further detailed with a unique identifier ("id"), a CWE hierarchical classification ("category"), a concise description ("title"), an in-depth explanation ("description"), an assessed impact level ("severity"), and the specific code location of the vulnerability ("location")

> **Answer to RQ1:** FORGE approach has built the most comprehensive smart contract vulnerability dataset compared to previous attempts, such as Smartbugs and DAppSCAN, in both scope and completeness.

4.2 RQ2: Performance of Information Extraction

To evaluate the performance of FORGE in information extraction, we first adopt a random sampling method based on confidence intervals from the dataset built in RQ1, following the practices of prior works [30, 80]. We set the confidence level to 95% and the confidence interval to 10. Using the calculation method implemented in [7], we randomly sampled 96 samples from the dataset. Two authors of this paper independently match the results produced by FORGE with original audit reports. Any discrepancies in this labeling were resolved through discussion with a third author.

Following the evaluation method in [14], we then calculate precision (P) and recall (R) for each entity type. Precision measures the accuracy of the extracted information, while recall measures the comprehensiveness of the extraction. These metrics are calculated by $\frac{\#\text{ correct entities extracted}}{\#\text{ entities extracted}}$ and $\frac{\#\text{ correct entities extracted}}{\#\text{ entities in ground truth}}$, respectively. The F1-score computes using the formula: $\frac{2\times(P\times R)}{P+R}$

Table 2: Information extraction performance of FORGE.

Entity	Precision(%)	Recall(%)	F1(%)
on-chain address	92.6	73.5	82.0
chain	100.0	82.3	90.3
URL	100.0	76.7	86.8
commit ID	100.0	78.9	88.2
vulnerability finding	91.7	73.2	81.4
severity	88.3	81.9	85.0
location	96.6	82.5	89.0
Average	95.6	78.4	**86.1**

Table 2 presents the information extraction performance of FORGE, structured with entity types divided into project metadata (on-chain address, chain, URL, commit ID) and vulnerability-related (vulnerability finding, severity, location), showing precision, recall, and F1-score for each category. We then compute FORGE's overall metrics as a weighted average across all entity types. **We find that FORGE achieves an overall precision of 95.6%, with a recall of 78.4%. The average F1-score (Macro-F1) is 86.1%.**

Our analysis of incorrect samples reveals several scenarios for improvement. Some reports present complex information (*e.g.* vulnerability codes) as images. In contrast, others employ checklist formats with icons or color highlights to mark inspection results, which poses challenges for single-modal models in detecting vulnerability information. These modality constraints impact extraction comprehensiveness but do not impact the reliability of successfully identified vulnerabilities.

> **Answer to RQ2:** Our FORGE framework achieved an overall F1 rate of 86.1% with a precision of 95.6%, highlighting its performance in extracting project and vulnerability information.

4.3 RQ3: FORGE's Classification v.s. Human Experts

To answer RQ3, we evaluate the consistency of FORGE's vulnerability classification against human expert judgments. Determining the category of a vulnerability with limited information is a subjective and experience-based task, which may lead to varying results even for professional security auditors [39]. For instance, the common smart contract vulnerability *Authorization through 'tx.origin'* has been categorized differently across security standards: SWC [70] categorized it under *CWE-477: Use of Obsolete Function* [48] due to its outdated and deprecated nature, while the EthTrust specification [52] classified it under *CWE-284: Improper Access Control* [43] because it can facilitate unauthorized access. Such discrepancies are particularly pronounced when analyzing vulnerabilities with limited contextual information. Thus, we employ Krippendorff's α (k-α) [1, 28], a robust reliability coefficient widely used in content analysis with multiple evaluators, to quantitatively assess the consistency between FORGE's classifications with human experts, instead of using human labeling results as ground truth.

Following the evaluation methodology from RQ2, we set a confidence level of 95% and a confidence interval of 10, randomly selecting 96 samples from the 27,497 vulnerabilities annotated with CWE categories. Two authors of this paper independently analyzed the description and code of each sample vulnerability, mapping it to the closest category in the CWE classification system based on its root cause. These ground truths were then compared with the ultimate classification results of the FORGE framework.

We calculated k-α using Equation 4, where D_o represents the observed disagreement, D_e denotes the expected disagreement by chance, n_{ijk} is the number of disagreements between coders i and j on coding unit k, d_{ijk} is the observed distance between coders i and j on coding unit k, and $E(d_{ijk})$ is the expected random disagreement distance.

$$\alpha = 1 - \frac{D_o}{D_e} = 1 - \frac{\sum_{i<j}\sum_k n_{ijk} d_{ijk}}{\sum_{i<j}\sum_k n_{ijk} E(d_{ijk})} \tag{4}$$

The resulting k-α coefficient is **0.87**, above the threshold of 0.80 suggested by Krippendorff [35] for drawing reliable conclusions. **This high level of agreement indicates that our LLM-based classification approach achieves a substantial level of reliability.**

There may be some inconsistencies in FORGE. We have open-sourced our dataset on GitHub to enable community engagement, allowing developers and researchers to report issues and contribute to its ongoing improvement. We analyzed the inconsistent results we found and discovered three main reasons causing these inconsistencies.

- **Some vulnerability descriptions are overly general**, requiring experience-based inference to accurately assign the corresponding CWE type, leading to inconsistency between human and LLM judgments.
- In **complex cases involving multiple vulnerability categories**, human experts tend to select more specific subcategories, while LLM tends to choose more generic parent categories.
- The **ambiguity of certain low-level CWE categories** introduces challenges for consistent classification. For example, distinguishing between *CWE-755: Improper Handling of Exceptional Conditions* [50] and *CWE-754: Improper Check for Unusual or Exceptional Conditions* [50] requires careful consideration due to their conceptual similarity.

> **Answer to RQ3:** FORGE achieved a 0.87 k-α coefficient, indicating substantial agreement between FORGE and human experts in CWE classification of smart contract vulnerabilities.

4.4 RQ4: Practicality of FORGE dataset

We assess the practicality of the FORGE dataset by evaluating how existing smart contract vulnerability detection tools perform when applied to our CWE-classified vulnerabilities derived from real-world audit reports. To conduct this assessment, we employ SmartBugs, a framework that integrates various tools to analyze smart contracts [18]. The selection criteria for the tools included the requirement that tools support source code as input, be automated, and be capable of detecting at least one type of CWE vulnerability. Based on these criteria, we selected 13 representative vulnerability detection tools listed in Table 3 with diverse detection techniques.

Two authors independently map the vulnerabilities each tool can detect to CWE categories, following the vulnerability mapping guidelines provided in SmartBugs-Wiki [66]. Any discrepancies in this mapping were resolved through discussion with a third author. Based on this mapping, we run the selected tools on the FORGE dataset with a default 300-second timeout and collect their detection results. We standardize evaluation at the contract level: if a tool detects any vulnerability within the contract that contains labeled vulnerabilities, we count it as a true positive (TP). We calculate precision, recall, and F1-score for each tool following the same methodology as [11, 87], as shown in Table 3.

Our experiments reveal that **the highest F1-score achieved by any tool was only 18.59%**, indicating a significant gap between current detection methods and the realities of real-world vulnerabilities. Notably, several tools, such as Manticore and Maian, failed to identify any true positives, which aligns with previous findings by Durieux *et al.* [18] and Sendner *et al.* [64]. This observation underscores the urgent need for more advanced detection tools and highlights the importance of utilizing large-scale, diverse datasets for accurate assessment and benchmarking.

Table 3: Analysis results of existing detection tools

Tool	TP	FP	FN	P(%)	R(%)	F(%)
Confuzzius [73]	13	462	2,250	2.74	0.57	0.95
Conkas [12]	2	51	677	3.77	0.29	0.55
Honeybadger [75]	0	7	52	0.00	0.00	0.00
Maian [53]	0	10	1,124	0.00	0.00	0.00
Manticore [51]	0	0	992	0.00	0.00	0.00
Mythril [13]	0	33	4,383	0.00	0.00	0.00
Osiris [74]	1	53	2,433	1.85	0.04	0.08
Oyente [34]	3	83	769	3.49	0.39	0.70
Securify [76]	1	3	1,004	25.00	0.10	0.19
Semgrep [63]	3,920	24,638	9,685	13.73	28.81	18.59
Slither [21]	4,016	40,468	14,936	9.03	21.19	12.66
Smartcheck [72]	3,939	34,446	10,512	10.26	27.26	14.91
Solhint [58]	3,271	19,976	11,485	14.07	22.17	17.21

Furthermore, **we observed a distinction between static analysis and symbolic execution approaches using our FORGE dataset**. Static analysis tools, *e.g.*, Semgrep, Slither, and Smartcheck, exhibited a higher number of true positives due to the lower environment requirements compared to symbolic execution tools like Oyente and Mythril. However, this advantage comes at the cost of a significantly higher false positive rate, leading to alarm fatigue that can hinder developers from addressing genuine vulnerabilities [16]. In contrast, symbolic execution tools, while more precise, suffered from timeouts on real-world complex contracts due to path explosion. These findings reveal gaps in existing detection approaches that might have been overlooked with smaller or less diverse datasets.

> **Answer to RQ4:** The FORGE dataset demonstrates high practicality by evaluating existing security tools, revealing significant limitations in current detection capabilities.

5 DISCUSSION

5.1 Implications

Our new FORGE dataset contains 27,497 real-world vulnerabilities from 6,454 audit reports, which facilitates a comprehensive characterization of smart contract vulnerabilities in real-world DApp projects.

5.1.1 Vulnerability Frequency vs. Severity. This study offers critical insights for security researchers and developers by examining the correlation between vulnerability frequency and severity, identifying high-impact vulnerabilities that warrant prioritized attention.

Smart contracts with the same vulnerability type can exhibit different potential impacts. For example, while smart contracts containing an *Integer Bug* may lead to serious financial loss, the vulnerability could be insignificant if it appears only in functions that are never invoked. To quantitatively represent the severity level of each vulnerability type, we follow the recommendations from previous work [62], adopting the Common Vulnerability Scoring System v4.0 (*CVSS v4.0*) [54], which is an open framework used to assess the severity of software vulnerabilities and provides a standardized way to rate them. We calculated the average *CVSS* score $\bar{s}(c)$ for each CWE category present in our dataset as follows:

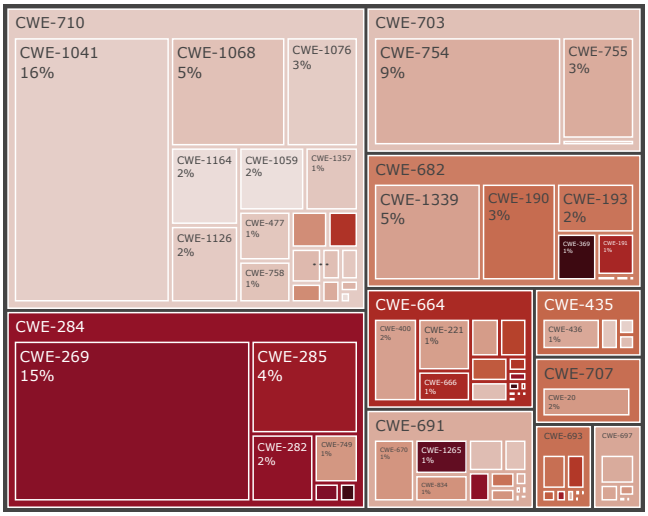

Figure 6: Risk priority visualization of smart contract from CWE perspective. The size of each rectangle represents the frequency of occurrence, while the color depth indicates the severity level (darker colors represent higher severity). [1]

$$\bar{s}(c) = \frac{1}{|\mathcal{V}_c|} \sum_{v \in \mathcal{V}_c} s(v) \tag{5}$$

$\mathcal{V}_c$ denotes the set of vulnerability findings in our dataset that belong to the CWE category c, with each vulnerability denoted as $v \in \mathcal{V}_c$. $s(v)$ represents the *CVSS* score of a vulnerability finding v. In our dataset, each vulnerability finding is assigned a severity level (i.e., `info`, `low`, `medium`, `high`, or `critical`) by auditors based on exploitation complexity and potential impacts. Each severity level corresponds to a *CVSS* score according to the *CVSS* standard. $|\mathcal{V}_c|$ denotes the cardinality of the set $\mathcal{V}_c$, *i.e.*, the total number of vulnerabilities in category c. This metric $\bar{s}(c)$ provides a quantitative representation of the average severity for each CWE category.

To visualize the distribution, we created a treemap in Figure 6 that incorporates CWE hierarchical information, illustrating both the frequency and severity of smart contract vulnerabilities from a CWE perspective. As shown in Figure 6, at the Pillar level, *CWE-710: Improper Adherence to Coding Standards* emerges as the most frequent root cause. *CWE-284: Improper Access Control* represents the vulnerability cause with the highest average severity. From a severity-frequency perspective, we observe that coding practice-related issues exhibit high frequency but relatively low severity. Conversely, there are distinct clusters of low-frequency but high-severity vulnerabilities, such as *CWE-369: Divide By Zero*.

> ***Finding 1:*** *High-severity smart contract vulnerabilities are not necessarily the most common.*

5.1.2 Academic Research Priorities vs. Actual Security Concerns. The left half of table 4 presents the top 10 specific CWE types in our dataset ranked by average *CVSS* score, where higher scores indicate a greater need for security audits in practical applications.

[1] A detailed interactive visualization is available in https://FORGE-security.github.io.

Table 4: FORGE average *CVSS* score *Top 10* vs. Academic Research Priority *Top 10*

#	CWE of FORGE Top 10	Detection Count Top 10 with CWE from Prior Research
1	CWE-940: Improper Verification of Source of a Communication Channel	Reentrancy (28) CWE-1265
2	CWE-369: Divide By Zero	Integer Bug (16) CWE-190/191
3	CWE-347: Improper Verification of Cryptographic Signature	Block-state Dependency (16) - CWE-829
4	CWE-1265: Unintended Reentrant Invocation of Non-reentrant Code Via Nested Calls	Control-flow Hijacking (15) CWE-691
5	CWE-287: Improper Authentication	Mishandled Exception (15) CWE-703
6	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	Assertion Failure (15) CWE-670
7	CWE-269: Improper Privilege Management	Ether Leakage (14) CWE-282
8	CWE-285: Improper Authorization	Suicidal Contract (14) CWE-749/826
9	CWE-282: Improper Ownership Management	Transaction Origin Use (12)- CWE-477/284
10	CWE-610: Externally Controlled Reference to a Resource in Another Sphere	Freezing Ether (11) CWE-684

To understand how existing research aligns with real-world security concerns, we compared these top 10 CWEs (called FORGE Top 10) shown in Table 4 with those most frequently targeted vulnerabilities in current smart contract analysis tools. According to the comprehensive survey conducted by Zhang *et al.* [85], the right half of table 4 presents the top 10 most frequently detected vulnerability types from 37 existing methods published on top-tier Software Engineering, Security, and Programming Language venues, which effectively representing the research community's primary focus.

Our comparison reveals a significant misalignment between academic research priorities and real-world security concerns. While existing tools heavily focus on well-known vulnerabilities such as reentrancy (28 tools), integer bugs (16 tools), and block-state dependency (16 tools), many of the most severe vulnerabilities identified in our dataset receive considerably less attention. For instance, *CWE-940* and *CWE-347*, which rank among the top 2 most severe vulnerabilities in our dataset, are rarely addressed by existing tools. Many high-severity smart contract vulnerabilities related to financial business logic identified from our dataset require complex semantic understanding and context awareness, making them challenging targets for automated analysis. This explains why current tools tend to focus on more structurally identifiable issues.

The misalignment highlights the importance of grounding security research in real-world data sources rather than focusing exclusively on machine-detectable vulnerabilities. Our findings suggest that the research community should recalibrate its focus to address the most severe security threats identified through actual security incidents and professional audits, even if these vulnerabilities present greater challenges for automated detection.

> ***Finding 2:*** *There exists a significant gap between the vulnerabilities prioritized by academic research and those that pose the greatest risks in practice.*

5.1.3 Smart Contract Vulnerabilities vs. Traditional Software Vulnerabilities. We conducted a comparative analysis between our smart contract vulnerability rankings and the widely recognized traditional software security risks. The *OWASP Top 10* represents a broad consensus on the most critical security risks to web applications

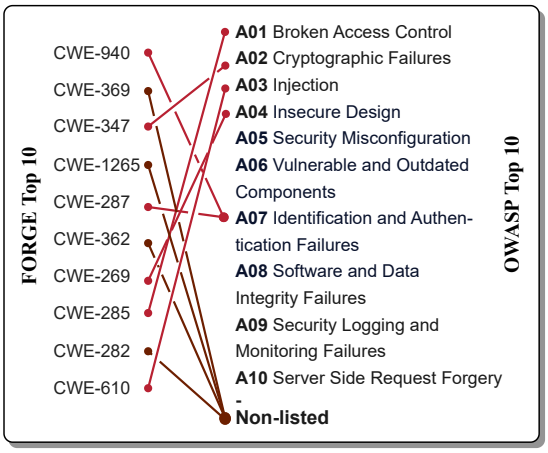

Figure 7: FORGE *Top 10* vs. *OWASP Top 10* [2]

in the traditional software domain [56]. *MITRE* provides a CWE perspective on the *OWASP Top 10*, mapping each risk to one or more CWE root causes [42]. To illustrate the divergence between the top 10 severity rankings from FORGE and *OWASP* root causes, we utilized a slope graph (Figure 7) that visually demonstrates the disparity in vulnerability rankings.

The comparison reveals significant differences between the vulnerability landscapes of smart contracts and traditional software applications. Notably, while some *OWASP* categories (A01-A04, A07) find corresponding instances in FORGE *Top 10*, they fail to encompass several of the most severe vulnerabilities in our dataset—including arithmetic issues, reentrancy problems, transaction order dependency (TOD), and ownership-related vulnerabilities that involve digital asset management and can lead to substantial losses if exploited. Additionally, A10 (Server-Side Request Forgery) does not have direct equivalents in our dataset.

This discrepancy between the FORGE *Top 10* and the *OWASP Top 10* highlights the need for tailored security practices in the smart contract domain. The wealth of experience and best practices accumulated in traditional vulnerability management, such as prioritizing the most critical security issues for efficient resource allocation and establishing standardized security assessment criteria, cannot be directly transferred to the smart contract ecosystem. The unique characteristics of smart contracts and the blockchain environment demand the development of specialized security frameworks, tools, and practices that address the specific challenges and risks associated with decentralized applications.

> ***Finding 3:*** *Our analysis reveals a significant divergence between traditional software security and smart contract vulnerabilities, which highlights the need for tailored security practices in the smart contract domain.*

5.1.4 For Ecosystem. Our study offers several important implications for the smart contract ecosystem: *(1) Research Focus Realignment.* Our dataset provides empirical evidence to guide research efforts toward high-severity vulnerabilities that currently receive insufficient attention, helping bridge the gap between academic research and practical security needs. *(2) Security-First Development.* The severity-ranked vulnerability distribution serves as a practical guideline for development practices, while our root cause analysis enables crucial feedback into the software development lifecycle for preventing entire vulnerability classes. *(3) Vulnerability Evolution Analysis.* By applying FORGE to audit reports across different time periods, researchers can track vulnerability pattern mutations, identify emerging threats, and understand the effectiveness of various security measures. *(4) Cross-Domain Knowledge Transfer.* By CWE classification, our dataset enables comparing security challenges between traditional applications and smart contract systems, facilitating knowledge transfer across domains.

[2] At the time of writing, the *OWASP Top 10* 2021 was the most recent version available. The figure presents qualitative comparison results; the scale may be stretched for visualization purposes.

5.2 Threats to Validity

Internal Validity. First, our dataset relies on LLM-based extraction and classification, which inherently introduces some imprecision when serving as ground truth datasets where high accuracy is crucial. This may impact practical utility. However, our evaluation in Section 4.2 demonstrates a high precision score for the generated entries, and our dataset is publicly accessible on GitHub, enabling community-driven validation and refinement through the issue tracking system. Second, our approach utilizes the CWE_s category, which was not originally designed for smart contract vulnerabilities. This potential mismatch could lead to inevitable inconsistencies in classification, as demonstrated in Section 4.3. Nevertheless, our evaluation analysis reveals that existing vulnerabilities from real-world audit reports have already been classified effectively into CWE_s, indicating sufficient coverage. Furthermore, our tree-of-thoughts method adaptively identifies the most appropriate vulnerability hierarchy level, mitigating potential inconsistencies.

External Validity. First, our dataset exclusively comprises audit reports, potentially introducing selection bias. However, this approach actually strengthens dataset authenticity since these reports from 47 renowned teams represent popular, real-world projects subjected to a professional security evaluation, thereby enhancing ecological validity rather than diminishing it. Second, our implementation employs LLama3 as the foundation model, raising concerns about how model performance might influence research conclusions, particularly with the emergence of new LLMs supporting longer context windows. This limitation is mitigated by our research objective, which focuses on dataset generation rather than context comprehension. Current LLMs, regardless of context length, still face fundamental challenges like "lost in the middle" effects. Within our framework, LLama3 demonstrates sufficient capability, and the modular design of FORGE allows straightforward model substitution when more effective alternatives become available.

6 RELATED WORK

Smart Contract Vulnerability Datasets. There has been substantial research effort in constructing high-quality smart contract vulnerability datasets. For example, Durieux *et al.* [18] introduced SmartBugs-wild, containing 47,398 solidity files from Ethereum with an average of only 204 lines of code per contract, and over

90% using an outdated 0.4+ compiler version. ScrawlD [82] incorporating vulnerability reports from multiple detection tools across 6,780 contracts. Both datasets were collected without manual vulnerability validation, including numerous false positives. Besides, their simplistic and outdated code samples fail to represent contemporary smart contract development practices. More recent efforts like Web3Bugs [85] and DAppSCAN [87] have attempted to provide higher-quality datasets by including deployed DApp audit reports. The larger one of these, DAppSCAN, covers 37 vulnerability types and includes 1,618 vulnerabilities. While existing datasets rely on manual annotation processes that constrain their scope and evolution, FORGE employs an automated approach, yielding a comprehensive dataset of 81,390 Solidity files with an average of 2,577 lines of code per project, encompassing 27,497 vulnerabilities across 296 CWE categories.

Vulnerability Classification. Decentralized Application Security Project Top 10 (DASP10) [22], Smart Contract Weakness Classification (SWC) [69] and Smart Contract Security Verification Standard (SCSVS) [62] have gained widespread adoption. However, these classifications often suffer from mixing different dimensions and providing non-orthogonal categories. Additionally, they have remained static for years, failing to evolve with the rapidly changing smart contract ecosystem [67]. In contrast, our work leverages the CWE classification, which provides a hierarchical, comprehensive, and regularly updated framework that encompasses both traditional software and smart contract vulnerability patterns.

Empirical Analysis of Real-World Smart Contract Vulnerabilities. Recent studies have conducted large-scale analyses of the smart contract security ecosystem. Zhang et al. [85] investigated 516 real-world vulnerabilities from 2021-2022, with a particular focus on machine-unauditable security bugs and their exploit patterns. Li et al.[29] evaluated 8 Static Application Security Testing (SAST) tools using a newly created taxonomy and benchmark (98.85% of vulnerabilities from previous work, 1.15% from manual audits of real BNB projects), providing guidance on SAST tool evaluation and selection. Sendner et al. [64] evaluated 18 vulnerability scanners across multiple datasets comprising over 4 million contracts, providing insights into the effectiveness of automated security tools. Compared to these works, our automated dataset construction approach enables more reliable and comprehensive empirical analysis through large-scale, real-world-sourced vulnerability findings. Moreover, from the unified perspective of the CWE, our work bridges the gap between smart contracts and traditional software, offering valuable comparative insights that were previously unavailable.

7 CONCLUSION

This paper presents FORGE, an automated framework for constructing smart contract vulnerability datasets from real-world audit reports. Our approach employs a divide-and-conquer strategy and tree-of-thoughts reasoning to extract and classify vulnerabilities into CWE categories, addressing critical limitations in manual dataset construction. In 229.5 hours, FORGE constructed a dataset containing 81,390 Solidity files with 27,497 vulnerabilities across 296 CWE categories, achieving expert-level classification consistency (k-α =0.87) and high extraction precision (95.6%). Our empirical analysis reveals distinct vulnerability patterns in smart contracts compared to traditional software, significant misalignment between research focus and real-world priorities, and limited effectiveness of existing security tools (max 18.59% F1). FORGE advances smart contract security through automated real-world vulnerability dataset construction with a unified CWE classification, establishing essential foundations for both academic research and industrial practice.

8 ACKNOWLEDGMENT

This project is supported by the Fundamental Research Funds for the Central Universities 226-2025-00004.

REFERENCES

[1] Toufique Ahmed, Premkumar Devanbu, Christoph Treude, and Michael Pradel. 2025. Can LLMs Replace Manual Annotation of Software Engineering Artifacts?. In *MSR '25: 22nd International Conference on Mining Software Repositories*.
[2] Meta AI. 2024. Introducing Meta Llama 3: The Most Capable Openly Available LLM to Date. https://ai.meta.com/blog/meta-llama-3/
[3] Andrei. 2024. llama-cpp-python. https://github.com/abetlen/llama-cpp-python/blob/main/llama_cpp/server/types.py#L25.
[4] Anthropic. 2024. Introducing the next Generation of Claude. https://www.anthropic.com/news/claude-3-family
[5] BscScan. 2025. BNB Smart Chain (BNB) Blockchain Explorer. https://bscscan.com/
[6] Vitalik Buterin. 2014. A Next-Generation Smart Contract and Decentralized Application Platform. *whitepaper* (2014), 3(37):2–1.
[7] Calculator.net. 2025. Sample Size Calculator. https://www.calculator.net/sample-size-calculator.html
[8] CertiK. 2025. CertiK - Hack3d: The Web3 Security Report 2024. https://certik.com/resources/blog/hack3d-the-web3-security-report-2024
[9] Chainsulting. 2020. *1inch v2 Audit Report.* Technical Report. Chainsulting. 16 pages.
[10] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Benjamin Livshits. 2024. Smart Contract and DeFi Security Tools: Do They Meet the Needs of Practitioners?. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, 1–13.
[11] Jiachi Chen, Zhenzhe Shao, Shuo Yang, Yiming Shen, Yanlin Wang, Ting Chen, Zhenyu Shan, and Zibin Zheng. 2025. NumScout: Unveiling Numerical Defects in Smart Contracts Using LLM-Pruning Symbolic Execution. *IEEE Trans. Softw. Eng.* 51, 5 (March 2025), 1538–1553. https://doi.org/10.1109/TSE.2025.3555622
[12] conkas. 2021. Nveloso/Conkas: Ethereum Virtual Machine (EVM) Bytecode or Solidity Smart Contract Static Analysis Tool Based on Symbolic Execution. https://github.com/nveloso/conkas
[13] Consensys. 2018. Consensys/Mythril: Mythril Is a Symbolic-Execution-Based Securty Analysis Tool for EVM Bytecode. It Detects Security Vulnerabilities in Smart Contracts Built for Ethereum and Other EVM-compatible Blockchains. https://github.com/Consensys/mythril
[14] John Dagdelen, Alexander Dunn, Sanghoon Lee, Nicholas Walker, Andrew S. Rosen, Gerbrand Ceder, Kristin A. Persson, and Anubhav Jain. 2024. Structured Information Extraction from Scientific Text with Large Language Models. *Nature Communications* 15, 1 (2024), 1418.
[15] DefiLlama. 2025. DefiLlama. https://defillama.com/
[16] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability Detection with Code Language Models: How Far Are We?. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 469–481.
[17] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Baobao Chang, Xu Sun, Lei Li, and Zhifang Sui. 2024. A Survey on In-context Learning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 1107–1128. https://doi.org/10.18653/v1/2024.emnlp-main.64
[18] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. ACM, 530–541.
[19] etherscan.io. 2024. Smart Contracts Audit. https://etherscan.io/directory/Smart_Contracts/Smart_Contracts_Audit_And_Security.
[20] etherscan.io. 2025. Ethereum (ETH) Blockchain Explorer. https://etherscan.io/
[21] Josselin Feist, Gustavo Greico, Alex Groce, and ACM. 2019. Slither: A Static Analysis Framework For Smart Contracts. In *2019 IEEE/ACM 2ND INTERNATIONAL WORKSHOP ON EMERGING TRENDS IN SOFTWARE ENGINEERING FOR BLOCKCHAIN (WETSEB 2019)*. 8–15.

[22] NCC Group. 2021. DASP - TOP 10. https://dasp.co/

[23] Zhuobing Han, Xiaohong Li, Hongtao Liu, Zhenchang Xing, and Zhiyong Feng. 2018. DeepWeak: Reasoning Common Software Weaknesses via Knowledge Graph Embedding. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 456–466.

[24] Hedera. 2025. What Is a Smart Contract Audit? https://hedera.com/learning/smart-contracts/smart-contract-audit

[25] Mingyuan Huang, Jiachi Chen, Zigui Jiang, and Zibin Zheng. 2024. Revealing Hidden Threats: An Empirical Study of Library Misuse in Smart Contracts. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, 1–12.

[26] Jinhao Jiang, Kun Zhou, Zican Dong, Keming Ye, Xin Zhao, and Ji-Rong Wen. 2023. StructGPT: A General Framework for Large Language Model to Reason over Structured Data. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 9237–9251.

[27] Takeshi Kojima, Shixiang (Shane) Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large Language Models Are Zero-Shot Reasoners. In *Advances in Neural Information Processing Systems*, Vol. 35. 22199–22213.

[28] Klaus Krippendorff. 2019. *Content Analysis: An Introduction to Its Methodology*. SAGE Publications, Inc.

[29] Kaixuan Li, Yue Xue, Sen Chen, Han Liu, Kairan Sun, Ming Hu, Haijun Wang, Yang Liu, and Yixiang Chen. 2024. Static Application Security Testing (SAST) Tools for Smart Contracts: How Far Are We?. In *Proceedings of the ACM on Software Engineering*, Vol. 1. 1447–1470. https://doi.org/10.1145/3660772

[30] Lu Liu, Lili Wei, Wuqi Zhang, Ming Wen, Yepang Liu, and Shing-Chi Cheung. 2022. Characterizing Transaction-Reverting Statements in Ethereum Smart Contracts. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE '21)*. IEEE Press, 630–641.

[31] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173.

[32] Peiyu Liu, Junming Liu, Lirong Fu, Kangjie Lu, Yifan Xia, Xuhong Zhang, Wenzhi Chen, Haiqin Weng, Shouling Ji, and Wenhai Wang. 2024. Exploring ChatGPT's Capabilities on Vulnerability Management. In *Proceedings of the 33rd USENIX Conference on Security Symposium (SEC '24)*. USENIX Association, 811–828.

[33] Xin Liu, Yuan Tan, Zhenghang Xiao, Jianwei Zhuge, and Rui Zhou. 2023. Not The End of Story: An Evaluation of ChatGPT-Driven Vulnerability Description Mappings. In *Findings of the Association for Computational Linguistics: ACL 2023*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, 3724–3731.

[34] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *CCS'16: 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 254–269.

[35] Giacomo Marzi, Marco Balzano, and Davide Marchiori. 2024. K-Alpha Calculator–Krippendorff's Alpha Calculator: A User-Friendly Tool for Computing Krippendorff's Alpha Inter-Rater Reliability Coefficient. *MethodsX* 12 (2024), 102545.

[36] Pascal C Meunier and Eugene H Spafford. 1999. Final Report of the 2nd Workshop on Research with Security Vulnerability Databases, January 1999. (1999).

[37] MITRE. 2024. 2024 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2024/2024_methodology.html

[38] MITRE. 2025. CWE - About CWE. https://cwe.mitre.org/about/index.html

[39] MITRE. 2025. CWE - CVE → CWE Mapping "Root Cause Mapping" Guidance. https://cwe.mitre.org/documents/cwe_usage/guidance.html.

[40] MITRE. 2025. CWE - CWE-1000: Research Concepts (4.16). https://cwe.mitre.org/data/definitions/1000.html

[41] MITRE. 2025. CWE - CWE-1192: Improper Identifier for IP Block Used in System-On-Chip (SOC) (4.16). https://cwe.mitre.org/data/definitions/1192.html.

[42] MITRE. 2025. CWE - CWE-1344: Weaknesses in OWASP Top Ten (2021) (4.16). https://cwe.mitre.org/data/definitions/1344.html.

[43] MITRE. 2025. CWE - CWE-284: Improper Access Control. https://cwe.mitre.org/data/definitions/284.html

[44] MITRE. 2025. CWE - CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html

[45] MITRE. 2025. CWE - CWE-295: Improper Certificate Validation. https://cwe.mitre.org/data/definitions/295.html

[46] MITRE. 2025. CWE - CWE-298: Improper Validation of Certificate Expiration. https://cwe.mitre.org/data/definitions/298.html

[47] MITRE. 2025. CWE - CWE-362: Concurrent Execution Using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html

[48] MITRE. 2025. CWE - CWE-477: Use of Obsolete Function. https://cwe.mitre.org/data/definitions/477.html

[49] MITRE. 2025. CWE - CWE-691: Insufficient Control Flow Management. https://cwe.mitre.org/data/definitions/691.html

[50] MITRE. 2025. CWE - CWE-755: Improper Handling of Exceptional Conditions. https://cwe.mitre.org/data/definitions/755.html

[51] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1186–1189.

[52] Chaals Nevile. 2023. EEA EthTrust Security Levels Specification V2. https://entethalliance.org/specs/ethtrust-sl/#sec-3-access-control

[53] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. Association for Computing Machinery, 653–663.

[54] NVD. 2025. NVD - Vulnerability Metrics. https://nvd.nist.gov/vuln-metrics/cvss

[55] OpenAI. 2023. GPT-4. https://openai.com/index/gpt-4/

[56] OWASP. 2024. OWASP Top Ten | OWASP Foundation. https://owasp.org/www-project-top-ten/

[57] Shengyi Pan, Lingfeng Bao, Xin Xia, David Lo, and Shanping Li. 2023. Fine-Grained Commit-Level Vulnerability Type Prediction by CWE Tree Structure. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, 957–969.

[58] Protofire. 2025. Solhint. https://github.com/protofire/solhint

[59] pymupdf. 2025. PyMuPDF. https://github.com/pymupdf/PyMuPDF

[60] Alexander Remie. 2024. *Uniswap v4 Core Security Assessment*. Technical Report. Trail of Bits.

[61] Claudia Ruggiero, Pietro Mazzini, Emilio Coppa, Simone Lenti, and Silvia Bonomi. 2024. SoK: A Unified Data Model for Smart Contract Vulnerability Taxonomies. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*. ACM, 1–13.

[62] Damian Rusinek and Paweł Kuryłowicz. 2021. Smart Contract Security Verification Standard. https://securing.github.io/SCSVS/

[63] Semgrep. 2025. Semgrep. https://semgrep.dev/p/smart-contracts

[64] Christoph Sendner, Lukas Petzi, Jasper Stang, and Alexandra Dmitrienko. 2024. Large-Scale Study of Vulnerability Scanners for Ethereum Smart Contracts. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2273–2290.

[65] Şevval Şimşek, Zhenpeng Shi, Howell Xia, David Sastre Medina, and David Starobinski. 2024. Poster: Analyzing and Correcting Inaccurate CVE-CWE Mappings in the National Vulnerability Database. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. Association for Computing Machinery, 5042–5044.

[66] SmartBugs. 2020. Vulnerabilities Mapping. https://github.com/smartbugs/smartbugs/wiki/Vulnerabilities-mapping

[67] Majd Soud, Grischa Liebel, and Mohammad Hamdaqa. 2023. A Fly in the Ointment: An Empirical Study on the Characteristics of Ethereum Smart Contracts Code Weaknesses and Vulnerabilities. *Empirical Software Engineering* 29, 1 (2023).

[68] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Yang Liu, and Yingjiu Li. 2024. LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning. arXiv:2401.16185

[69] SWC. 2020. Smart Contract Weakness Classification (SWC). https://swcregistry.io/

[70] SWC. 2020. SWC-115 - Smart Contract Weakness Classification (SWC). https://swcregistry.io/docs/SWC-115/

[71] Zhen Tan, Dawei Li, Song Wang, Alimohammad Beigi, Bohan Jiang, Amrita Bhattacharjee, Mansooreh Karami, Jundong Li, Lu Cheng, and Huan Liu. 2024. Large Language Models for Data Annotation and Synthesis: A Survey. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, 930–957.

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

[73] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. 103–119.

[74] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. Association for Computing Machinery, 664–676.

[75] Christof Ferreira Torres, Mathis Steichen, and Radu State. 2019. The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts. In *Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19)*. USENIX Association, 1591–1607.

[76] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. Association for Computing Machinery, 67–82.

[77] Uniswap. 2024. Uniswap/v4-Core at Audit/Trail-of-Bits. https://github.com/Uniswap/v4-core/tree/audit/trail-of-bits

[78] Zhiyuan Wan, Xin Xia, David Lo, Jiachi Chen, Xiapu Luo, and Xiaohu Yang. 2021. Smart Contract Security: A Practitioners' Perspective. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*. IEEE Press, 1410–1422.

[79] Jiaan Wang, Yunlong Liang, Fandong Meng, Zengkui Sun, Haoxiang Shi, Zhixu Li, Jinan Xu, Jianfeng Qu, and Jie Zhou. 2023. Is ChatGPT a Good NLG Evaluator? A Preliminary Study. In *Proceedings of the 4th New Frontiers in Summarization Workshop*, Yue Dong, Wen Xiao, Lu Wang, Fei Liu, and Giuseppe Carenini (Eds.). Association for Computational Linguistics, 1–11.

[80] Shuo Yang, Jiachi Chen, and Zibin Zheng. 2023. Definition and Detection of Defects in NFT Smart Contracts. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, 373–384.

[81] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 11809–11822.

[82] Chavhan Sujeet Yashavant. 2021. Sujeetc/ScrawlD.

[83] Jiashuo Zhang, Jiachi Chen, Zhiyuan Wan, Ting Chen, Jianbo Gao, and Zhong Chen. 2024. When Contracts Meets Crypto: Exploring Developers' Struggles with Ethereum Cryptographic APIs. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, 1–13.

[84] Jiashuo Zhang, Yiming Shen, Jiachi Chen, Jianzhong Su, Yanlin Wang, Ting Chen, Jianbo Gao, and Zhong Chen. 2024. Demystifying and Detecting Cryptographic Defects in Ethereum Smart Contracts. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 114–126.

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

[86] Zibin Zheng, Kaiwen Ning, Qingyuan Zhong, Jiachi Chen, Wenqing Chen, Lianghong Guo, Weicheng Wang, and Yanlin Wang. 2024. Towards an Understanding of Large Language Models in Software Engineering Tasks. *Empirical Software Engineering* 30, 2 (2024), 50.

[87] Zibin Zheng, Jianzhong Su, Jiachi Chen, David Lo, Zhijie Zhong, and Mingxi Ye. 2024. DAppSCAN: Building Large-Scale Datasets for Smart Contract Weaknesses in DApp Projects. *IEEE Trans. Softw. Eng.* 50, 6 (2024), 1360–1373.

[88] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. 2020. An Overview on Smart Contracts: Challenges, Advances and Platforms. *Future Generation Computer Systems* 105 (2020), 475–491.

[89] Haozhe Zhou, Amin Milani Fard, and Adetokunbo Makanju. 2022. The State of Ethereum Smart Contracts Security: Vulnerabilities, Countermeasures, and Tool Support. *Journal of Cybersecurity and Privacy* 2, 2 (2022), 358–378.

[90] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. 2023. SoK: Decentralized Finance (DeFi) Attacks. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2444–2461.