A Curated Solidity Smart Contracts Repository of Metrics and Vulnerability

Giacomo Ibba
University of Cagliari
Cagliari, Italy
g.ibba14@studenti.unica.it

Sabrina Aufiero
University College London
London, United Kingdom
sabrina.aufiero.22@ucl.ac.uk

Rumyana Neykova
Brunel University
London, United Kingdom
rumyana.neykova@brunel.ac.uk

Silvia Bartolucci
University College London
London, United Kingdom
s.bartolucci@ucl.ac.uk

Marco Ortu
University of Cagliari
Cagliari, Italy
marco.ortu@unica.it

Roberto Tonelli
University of Cagliari
Cagliari, Italy
roberto.tonelli@unica.it

Giuseppe Destefanis
Brunel University
London, United Kingdom
giuseppe.destefanis@brunel.ac.uk

ABSTRACT

Smart contracts (SCs) significance and popularity increased exponentially with the escalation of decentralised applications (dApps), which revolutionised programming paradigms where network controls rest within a central authority. Since SCs constitute the core of such applications, developing and deploying contracts without vulnerability issues become key to improve dApps robustness to external attacks. This paper introduces a dataset that combines smart contract metrics with vulnerability data identified using Slither, a leading static analysis tool proficient in detecting a wide spectrum of vulnerabilities. Our primary goal is to provide a resource for the community that supports exploratory analysis, such as investigating the relationship between contract metrics and vulnerability occurrences. Further, we discuss the potential of this dataset for the development and validation of predictive models aimed at identifying vulnerabilities, thereby contributing to the enhancement of smart contract security. Through this dataset, we invite researchers and practitioners to study the dynamics of smart contract vulnerabilities, fostering advancements in detection methods and ultimately, fortifying the resilience of smart contracts.

CCS CONCEPTS

• **Theory of computation** → **Data modeling**; • **Security and privacy** → *Vulnerability management*; *Software security engineering*; • **General and reference** → Metrics.

KEYWORDS

Smart Contracts, Ethereum, Blockchain, Vulnerability Detection, Software Engineering, Data Analysis

PROMISE '24, July 16, 2024, Porto de Galinhas, Brazil
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0675-2/24/07
https://doi.org/10.1145/3663533.3664039

ACM Reference Format:
Giacomo Ibba, Sabrina Aufiero, Rumyana Neykova, Silvia Bartolucci, Marco Ortu, Roberto Tonelli, and Giuseppe Destefanis. 2024. 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 (PROMISE '24), July 16, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3663533.3664039

1 INTRODUCTION

Ethereum smart contracts have emerged as a foundational technology in the development of decentralised applications (dApps), enabling secure and automated transactions without the need for central oversight. These programs are critical in shifting the control mechanisms of various applications from centralised entities to distributed networks. Beyond financial transactions, smart contracts are central in a diverse range of applications including notary services, gaming platforms, and sectors demanding high reliability and swift response times, such as IoT and security systems. This broad applicability shows the transformative potential of smart contracts across different domains, highlighting the need for rigorous analysis and understanding of their structure, functionality, and security challenges.

As integral components of dApps, smart contracts must be deployed with a high degree of security assurance. Their role is central in safeguarding the entire dApp ecosystem from potential threats. Vulnerabilities in a single smart contract can have a cascading effect, jeopardising the security of interconnected components and potentially leading to severe consequences such as unauthorised access, data breaches, and substantial financial losses.

The challenge in achieving secure smart contracts is amplified by the inherent complexities of dApps. These applications often operate in a highly interconnected environment, with dependencies that extend beyond their immediate ecosystem to include external libraries and services. This complexity is further exacerbated by the fast paced evolution of blockchain technology, where new patterns and practices emerge rapidly, making it difficult to maintain an up-to-date understanding of security best practices.

The decentralised nature of these applications means that traditional centralised security controls are not applicable, requiring developers to adopt new paradigms of security thinking. As dApps become more sophisticated, the potential impact of security vulnerabilities grows. Keeping abreast of vulnerabilities and employing comprehensive analysis and mitigation techniques are essential steps in ensuring the resilience and reliability of dApps against the threats they face.

This data paper introduces a dataset consisting of a selected collection of Solidity smart contracts containing around 50k contracts (which we are continuously expanding), each accompanied by vulnerability reports generated by Slither [5]. Slither is renowned in the software development community for its effectiveness as a static analysis tool, capable of identifying a wide set of vulnerabilities within smart contract code. Our dataset goes beyond mere vulnerability identification; it offers an extensive analysis of both contract-level and function-level metrics. These metrics provide insights into the structural and operational characteristics of smart contracts, enabling a deeper understanding of the factors that may influence their security posture.

With this dataset we provide a tool for both researchers and developers focused on Ethereum smart contracts. The dataset's rich detail on vulnerabilities and contract metrics directly supports the development of more secure coding practices and the improvement of security analysis tools. It enables detailed study into how contract characteristics correlate with vulnerabilities, paving the way for advanced vulnerability prediction models. Developers can use this dataset to test and refine their security approaches, aiming to decrease the occurrence of flaws in deployed contracts.

With this dataset, publicly available on GitHub[1], we promote a collaborative approach to tackling smart contract security challenges. It encourages input from a wide range of contributors, including academics, blockchain developers, and security professionals, to work together in addressing smart contract vulnerabilities.

The structure of the paper is as follows: Section 2, details the motivations driving this research and the creation of the dataset and discusses related work, analysing existing smart contracts (SCs) datasets, vulnerability detection, testing tools, and tools for analysing decentralised applications (dApps). Section 3 outlines the methodology used to collect data, extract software metrics from SC source code, and conduct the vulnerability detection process. The composition and statistical analysis of the dataset are described in Section 4, while Section 5 offers insights into software metrics and their distribution across different categories and impacts. Potential applications of the dataset are explored in Section 6. Section 7 discusses the threats to the validity of our research, and the paper concludes with Section 8 and Section 9, highlighting future research directions and summarising the content of the study.

2 MOTIVATION AND RELATED WORK

The adoption of smart contracts across various sectors shows their impact on digital transactions and decentralised applications, but this rapid integration also brings to light significant security vulnerabilities that pose risks to users and the integrity of the blockchain network. Despite the advancements in development practices and tools aimed at improve security, the persistent emergence of vulnerabilities in smart contracts reveals gaps in current methodologies and the need for new solutions. The motivation behind this work stems from the critical need to address these security challenges head-on. By providing a large dataset of Solidity smart contracts, complete with vulnerability reports and metrics at contract and function level, we aim to bridge these gaps. This dataset represents a collection of structural and functional aspects of smart contracts, offering insights into their security dynamics. The dataset introduced in this paper is designed to help researchers, developers, and security experts to uncover patterns, develop predictive models, with the goal of obtaining more resilient smart contracts.

Here, we explore the literature to present an overview of the current state of resources and tools available for smart contract analysis. Our review covers three main areas: the availability and scope of smart contracts datasets, the range and effectiveness of vulnerability detection tools, and the frameworks developed for analysing decentralised applications (dApps) and smart contracts.

2.1 Smart Contracts Datasets

Among the smart contracts datasets, SmartCorpus [14] stands out as an organised repository that provides access to Solidity source code, along with relevant metadata such as bytecode and application binary interfaces for Ethereum smart contracts. This repository facilitates easy and systematic retrieval of contract information. Additionally, SmartCorpus includes software metrics for each contract, obtained through the PASO framework [1], which supports the extraction of both object-oriented and Solidity-specific metrics. Similarly, our dataset provides software metrics derived from smart contract source codes. Yet, it goes a step further by incorporating vulnerability reports produced by Slither. This addition opens avenues for deeper analysis, specifically enabling the exploration of potential links between software metrics and the presence of vulnerabilities. The last update of SmartCorpus dates back to 2022 and encompasses around 35k addresses of verified contracts, while our repository provides approximately 50k contracts, and is continuously expanding.

SmartBugs offers a dual resource: an analytical framework for smart contracts (SC) and repositories containing Solidity source codes. Its curated selection features 143 contracts[2] identified with specific vulnerability issues. Meanwhile, the SmartBugs wild dataset [4] compiles a collection of 47,398 contracts from the Ethereum network. The Skelcodes dataset [3], on the other hand, provides both the deployment and runtime code of Ethereum main chain contracts, including those that have been self-destructed, alongside the application binary interface (ABI) for each.

Our dataset advances beyond these collections by incorporating software metrics directly extracted from the contracts' source codes, an aspect not covered by SmartBugs or Skelcodes. In addition to these metrics, our dataset offers vulnerability reports for the contracts it includes, providing a more comprehensive tool for SC analysis and research.

The Smart Contract Sanctuary repository [12] serves as another significant source of smart contracts, extracting Solidity SC not only from Ethereum's main net but also from test nets, along with

[1] https://github.com/giacomofi/SmarthER

[2] https://github.com/smartbugs/smartbugs-curated

contracts from other blockchain platforms like Polygon, Tron, and Avalanche. Despite its broad scope and being the most complete in terms of the sheer volume of smart contract samples, the Smart Contract Sanctuary lacks detailed artifacts and metadata that accompany these contracts. Building on the foundation provided by the Smart Contract Sanctuary, our dataset significantly extends its value. We have added software metrics and vulnerability reports for the contracts derived from this repository. This enhancement introduces a deeper layer of analysis and insight, making it a more potent tool for researchers and practitioners interested in the security and performance of smart contracts.

2.2 Vulnerability Detection Tools

Throughout Ethereum's development, a variety of tools for detecting vulnerabilities have been introduced. Oyente [9], an early tool, uses symbolic execution compatible with Ethereum Virtual Machine (EVM) bytecode to identify smart contract bugs.

Osiris [16] focuses on finding arithmetic bugs in smart contracts by using symbolic execution and taint analysis, expanding on Oyente's approach.

Securify [17] analyses contracts' dependency graphs through symbolic analysis to extract semantic insights, checking for compliance and violation patterns critical for property validation.

Slither [5], through static analysis, examines smart contract code to provide feedback aimed at improving code structure. It analyses source code and binaries without code execution.

SmartCheck [15] translates Solidity code into XML-based intermediate representations for verification against XPath patterns, also using static analysis.

Mythril, a tool designed for EVM bytecode, excels in finding security vulnerabilities across a range of EVM-compatible blockchains by employing symbolic execution, SMT solving, and taint analysis.

Additional tools like Manticore [10], Ethainter [2], and Sfuzz [11] offer various approaches to vulnerability detection and testing. Despite the diversity of tools, Slither is particularly noted for balancing performance, speed, and the breadth and depth of vulnerability detection.

.

3 RESEARCH METHODOLOGY

This section outlines the methodological approach we used to compile our dataset, as presented by the workflow in Figure 1.

The figure provides a visual representation of the steps involved in assembling the dataset.

The process begins with the application of a Python script that triggers the execution of Slither, which scans the smart contracts, generating a detailed vulnerability report for each contract. Concurrently, our custom-built tool extracts a set of software metrics for these contracts. These vulnerability reports are then stored as JSON files, and the software metrics are compiled into CSV files, facilitating easy access and analysis.

The end of this process is the integration of the source codes, vulnerability reports, and software metrics into a unified repository. This repository serves as the foundation for subsequent analysis and research. The following subsections will present each stage of the methodology in detail, providing insight into the development and application of the tools used for extracting software metrics and vulnerability reports.

3.1 Data Collection

For our dataset, we chose to use existing open-source repositories of smart contracts. Our selection was the Smart Sanctuary dataset[3] [12], compared with other options like SmartBugs Wild, Skelcodes, and SmartCorpus. This decision was based on several factors. The Smart Sanctuary dataset is extensive, containing over 7 GB of data from Ethereum main net contracts alone, and includes a broad range of pragma versions used in smart contracts. Its last update was on July 13, 2023, which means it contains the most current versions of Solidity at the time of our research. In contrast, the SmartBugs Wild dataset's latest update was in January 2020, missing newer pragma versions and the latest contract design patterns that could be valuable for our study.

While the Skelcodes dataset is up-to-date, its source codes were no longer available in the repository, requiring additional steps to obtain the source code from Etherscan[4]. Given these considerations, the SmartSanctuary dataset was the most practical choice, providing a solid foundation for our dataset with its recent updates and wide coverage of contract versions.

3.2 Metrics Extraction

The extraction of software metrics from smart contracts within our dataset is achieved through the integration of two distinct tools: Slither [5] (which execution is triggered through a Python script) and MindTheDapp [6]. Slither is a well-established tool and provides an extensive set of code metrics. MindTheDapp introduces the capability to define and compute a specialized metric, "Coupling Between Contracts." This is facilitated by leveraging ANTLR (Another Tool for Language Recognition) to create a parser based on the Solidity grammar[5]. By merging the capabilities of Slither and MindTheDapp, our dataset includes both standard code metrics and assessments of contract interactions.

To ensure the reliability of our metric extraction process, we conducted manual code inspection, unit and integration testing, and manual reviews of a subset of contracts. These preliminary tests were aimed at verifying the accuracy of the extracted metrics.

One drawback in using Slither is its reliance on the AST (Abstract Syntax Tree) generated by the Solidity compiler, and hence its usage requires compiling the contracts. Compiling a large dataset of smart contract codebases is a challenging endeavour. The process of accurately identifying and integrating dApps dependencies is complex and time-consuming. While manually resolving dependencies might be feasible for analyzing a single dApp, this approach becomes impractical for large-scale datasets of smart contracts. The difficulty is particularly pronounced with contracts that incorporate external libraries and modules. To address this issue and automate the installation and resolution of dependencies, we leverage the DAI toolset [7], which is designed to manage these challenges across large datasets of smart contracts.

[3] https://github.com/tintinweb/smart-contract-sanctuary
[4] https://etherscan.io/
[5] https://github.com/solidityj/solidity-antlr4/blob/master/Solidity.g4

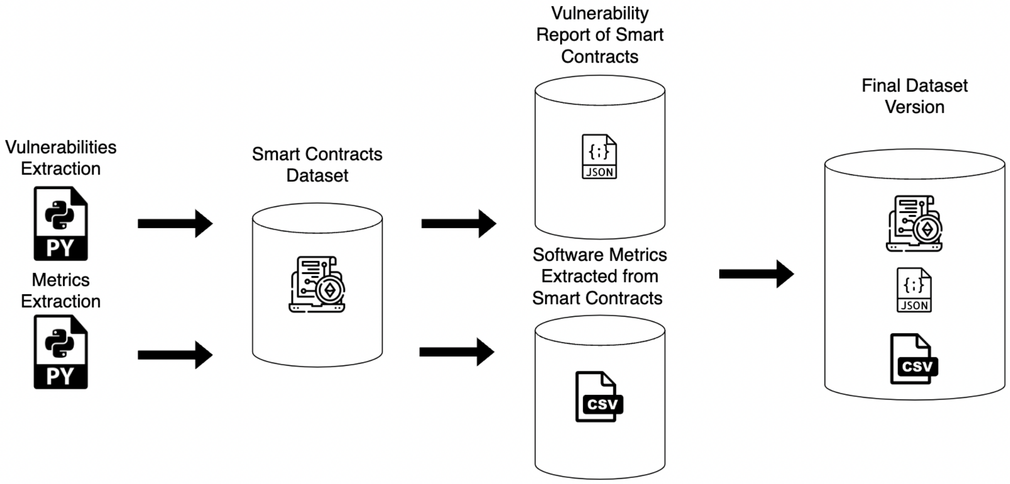
Vulnerabilities Extraction
PY
Metrics Extraction
PY
Smart Contracts Dataset
Vulnerability Report of Smart Contracts
JSON
Software Metrics Extracted from Smart Contracts
CSV
Final Dataset Version
JSON
CSV

Figure 1: Toolchain employed to build the dataset

Additionally, to compile a smart contract correctly, the specific version of the Solidity compiler, as indicated by the contract's pragma directive, must be used. To manage the variety of Solidity contracts and their respective compiler version requirements, a dynamic approach is essential. This involves the manual installation of all available versions of the Solidity compiler and using the *solc-select*[6] tool to change the compiler version as needed when processing the dataset.

This strategy allows for each smart contract to be compiled as closely as possible to its original development specifications, ensuring the accuracy of the metrics extracted from these contracts.

The following general cases provide insights into this choice:

- pragma solidity ˆ0.5.0;: The file employing this pragma version will compile with versions greater than 0.5.0, but it will not be compatible with versions greater starting from 0.6.0, and lower than 0.5.0.
- pragma solidity =0.5.0;: The file employing this pragma version will compile exclusively if the selected compiler version is 0.5.0.
- pragma solidity ~0.5.0;: The file employing this pragma version is compatible with versions aligning with 0.5.0, including minor updates and bug fixes. Solidity compiler versions in which crucial updates have been introduced are incompatible.
- pragma solidity >0.5.0 <0.7.0;: The file employing this pragma version is compatible with versions aligning between 0.5.0 and 0.7.0. This presents a unique challenge due to the non-trivial nature of precisely determining the matching version alignment between the compiler and pragma directive. Files encompassing this pragma necessitate a thorough examination of smart contract functionalities to ascertain the precise compatible version.

From each SC the following contract and function level metrics are extracted:

- **No. of Raw Lines**: The number of lines of code (LOC) of the SC.
- **No. of Contracts**: The number of contracts defined inside the SC.
- **Inheritance Depth**: This metric measures the layers of inheritance of a specific contract. A higher depth could indicate a more complex contract structure.
- **Coupling Between Contracts**: Indicates the number of other contracts or libraries that a contract interacts with. Higher coupling may lead to increased complexity and potential risks.
- **State Variable Count**: Represents the number of state variables in a contract. A higher count could lead to more complex contract interactions.
- **Avg Variable Count**: The average number of local variables used across all functions in a contract. This can be an indicator of how much temporary storage a contract uses.
- **Max Local Variable**: The maximum number of local variables used in any single function within a contract.
- **Number of Functions**: The total number of functions in a contract. This metric gives an idea of the contract's functionality and complexity.
- **Number of Parameters**: The number of parameters given as input to functions. A higher number could make the function more complex and harder to use.
- **Nesting Depth**: Represents the depth of nested loops and conditionals within a function. Deeper nesting can make a function harder to understand and maintain.
- **Function Calls**: Counts the number of times a function calls other functions. Frequent calls can lead to intricate function behaviours and interactions.

[6] https://github.com/crytic/solc-select

- **Cyclomatic Complexity**: Measures the number of linearly independent paths through a function's source code. Higher values denote more complex functions.
- **Local Variable Count**: Indicates the number of local variables within a function. A higher count could imply more complex computations and logic within the function.

In our catalog, we introduced a new metric: "Coupling Between Contracts" (CBC). As briefly explained above, this metric quantifies the number of distinct contracts or libraries that a given smart contract directly interacts with. A higher CBC value suggests a greater level of interdependence between contracts, which, while indicative of a potentially rich set of functionalities, also flags an increase in complexity and potential security vulnerabilities.

Comparatively, the traditional software engineering metric, Coupling Between Objects (CBO), measures the degree to which different classes are interdependent within object-oriented programming. While CBO provides insights into the complexity and maintainability of software systems by highlighting the tightness of coupling between classes, CBC is tailored to capture the dynamics of smart contract ecosystems. Unlike CBO, which focuses on class-level interactions within a single software system, CBC specifically addresses the interactions across the decentralised and distributed nature of blockchain applications. This distinction is important in the context of smart contracts, where interactions are not limited to internal components but extend to other contracts and libraries within the blockchain network. By measuring CBC, developers and analysts can gain an understanding of the contract's integration within the broader ecosystem, potentially identifying areas where reduced coupling could mitigate risks and simplify the system.

The extraction of these software metrics could contribute to giving preliminary insights about dApps complexity and development practices [8]. Moreover, studying the correlation between the exposures detected by Slither and the extracted metrics could give valuable insights and suggestions on SC's security improvement.

3.3 Exposures Analysis

The tool generates vulnerability reports by conducting static analysis with Slither, which is also a key part of the tool used for extracting metrics. The structure of this tool, includes components that are similar to those used in the metrics extraction setup. This setup allows the tool to efficiently search through smart contract code for patterns that could indicate security vulnerabilities, all without needing to run the contract.

Our focus here is specifically on the pragma version of Solidity defined in the smart contracts we are scanning. We have already covered the challenges of identifying the exact pragma version and potential complications in compiling smart contracts. The key distinction with the tool mentioned earlier is its use of Slither. While we use Slither in the metrics extraction tool to gather basic components of the smart contract, here it is used differently. In this stage, Slither scans the contract to create a detailed report of any vulnerabilities it finds. It classifies vulnerabilities into five categories based on their potential impact:

- **High**: this level signifies critical vulnerabilities that could lead to severe outcomes, including financial loss or data breaches.
- **Medium**: vulnerabilities at this level might affect how well different parts of a contract work together or could restrict what the contract can do.
- **Low**: these are issues that might cause the contract to behave in unexpected ways, though they do not prevent the contract's normal operation.
- **Informational**: here, the tool flags up minor concerns, such as unused functions, that do not directly affect security or performance.
- **Optimisation**: suggestions for improving contract efficiency, like reducing gas costs, fall into this category.

The findings generated by Slither are serialised into a JSON file format, a measure undertaken to enhance readability and expedite access to pertinent insights. Particularly, we are interested in:

- **Type**: provides the type of construct exposed to the vulnerability issue. The construct could either be a contract, a function, a variable, an arithmetic operation or function call (called nodes in the Slither report), or even the pragma version.
- **Name**: it represents the name of the construct exposed to the vulnerability issue. If the structural element labeled as 'type' aligns with 'node', the complete line of code is documented and reported.
- **Is Dependency**: a boolean value stating if the construct is whether or not a dependency.
- **Lines**: the precise line number or line numbers of code of the vulnerable pattern.
- **Description**: the description of the exposure detected by Slither.
- **Check**: the type of issue detected by Slither. A wide spectrum of vulnerability issues is covered, which can be examined in the detector documentation[7].
- **Impact**: the severity level of the exposure.
- **Confidence**: the level of confidence Slither has in detecting the specific issue.

4 DATASET DESCRIPTION AND DISCUSSION

Figure 2 shows the structure of our dataset repository. The **main net** directory contains the source code of 47932 contracts sourced from the smart sanctuary dataset. In this directory, smart contracts are organised into subfolders named after the initial characters of their associated addresses. For instance, contracts whose addresses begin with "00" are placed in the "00" folder, and those starting with "0xff" are found in the "ff" folder.

The **report** directory is divided into four directories, each mirroring the structure of the main net directory to organise the vulnerability reports. Slither generates these reports for every contract from the main net directory. Following the same organisational pattern, vulnerability reports for contracts starting with "0x00" are located in the "00" folder. Importantly, these folders also include reports for contracts that failed to compile, making it easier to track errors and improve our analysis.

The **software metrics** directory houses the metrics for each contract from the main net directory. For every subfolder in *main net*, corresponding CSV files compile the extracted metrics from

[7] https://github.com/crytic/slither/wiki/Detector-Documentation

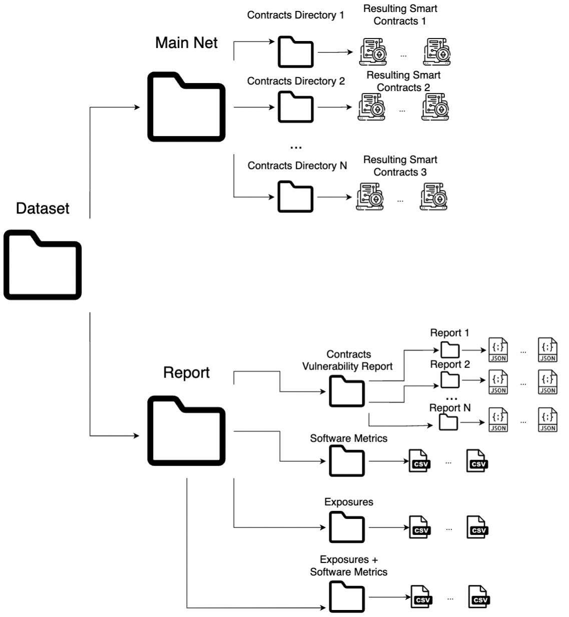

Figure 2: Visual representation of dataset organisation

the contracts within that folder. For example, the *00metrics.csv* file contains all the metrics from contracts in the "00" folder. The specifics of these software metrics, including their definitions and significance, are detailed in Section 3.2.

The **exposure** directory aggregates the vulnerability report data into CSV files, making it easy to access information on vulnerabilities related to functions and contracts quickly. This directory's layout is designed to reflect the organisation seen in the *metrics* folder. For instance, the *00VulnerabilityReport.csv* file summarises the vulnerabilities for contracts located in the "00" folder. Details on the types of data included in these CSV files, such as specific vulnerability attributes, are further discussed in Section 3.3.

The final sub-directory under the *report* main directory features CSV files that combine data from both metrics extraction and vulnerability analysis. Each file in this sub-directory lists all metrics for functions or contracts, along with any identified vulnerabilities, using information outlined in the exposure CSV files. For functions or contracts found to be without vulnerabilities, the relevant columns in these comprehensive CSV files are filled with '–' to indicate the absence of vulnerabilities.

The vulnerability report counts:

- 53335 informational reports.
- 1443 optimisation suggestions.
- 38673 low impact vulnerabilities.
- 11794 medium impact vulnerabilities.
- 4254 high impact vulnerabilities.

Table 1 reports the high-impact vulnerability issues exposing the contracts encompassed in our dataset. Reviewing the data in Table 1, we notice a significant number of vulnerabilities are related to reentrancy, known for their potential to cause irreversible ETH losses. The dataset also reveals issues like unauthorised Ether transfers to generic addresses, unchecked transfer call outcomes, and uninitialised state variables. Except for the uninitialised state variable vulnerabilities, which Slither detects reliably, the most frequently found vulnerabilities are identified with medium confidence. This observation points to the need for careful scrutiny to discern possible false positives among these findings. Nonetheless, it is fundamental to recognise that even a few high-severity vulnerabilities can have devastating effects. For example, reentrancy issues can allow attackers to repeatedly execute external calls, manipulating currency transfer logic to drain a contract's funds entirely.

Table 1: Vulnerability Samples

Vulnerability	Number of Samples	Confidence
reentrancy-eth	2012	Medium
uninitialized-state	419	High
unchecked-transfer	660	Medium
arbitrary-send-eth	614	Medium
arbitrary-send-erc20-permit	5	Medium
arbitrary-send-erc20	109	High
weak-prng	131	Medium
controlled-delegatecall	49	Medium
incorrect-shift	141	High
controlled-array-length	43	Medium
msg-value-loop	8	Medium
suicidal	6	High
shadowing-state	24	High
unprotected-upgrade	30	High
delegatecall-loop	3	Medium

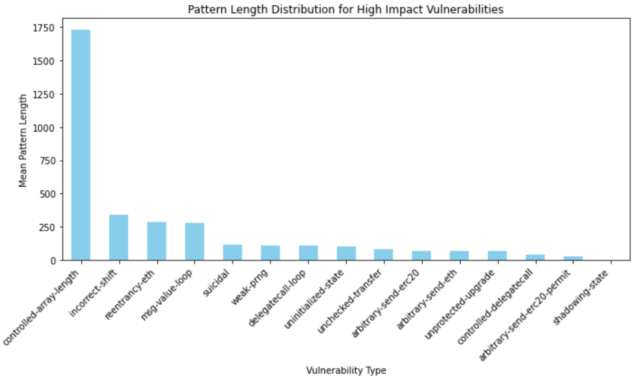

Figure 3: Pattern Length Distribution for High Impact Vulnerabilities

Figure 3 shows how average pattern lengths vary among high-impact vulnerabilities identified in our dataset. Vulnerabilities such as 'controlled-array-length', 'incorrect-shift', 'reentrancy-eth', 'msg value loop', and 'suicidal' stand out due to their longer code patterns compared to other high-impact vulnerabilities. This suggests that these types are associated with more complex or extensive code segments within smart contracts, highlighting regions that may be particularly vulnerable to security breaches. The extended pattern lengths of these vulnerabilities underline the importance of scrutinising these areas closely for potential risks.

5 QUALITATIVE ANALYSIS

For presenting a preliminary qualitative analysis, we selected the "0aFullReport.csv" file in the dataset, without specific preference for its content over others. The first step in our analysis involved processing this file to extract a unique list of contracts and their associated metrics, ensuring that each contract was represented only once to avoid duplication. This process was key for setting a clear foundation for subsequent visual examination and analysis of the metrics associated with each contract. The second step focused on a set of predefined metrics for assessing the structural complexity and design aspects of smart contracts. These metrics included Inheritance Depth, Coupling Between Contracts (CBC), State Variable Count, Average Local Variables, Maximum Local Variables, and the Number of Functions.

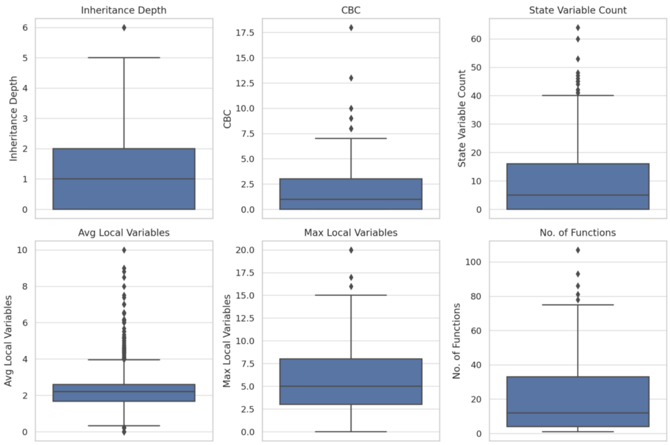

Figure 4: Distribution of Smart Contract Metrics

Figure 4 offers a detailed visual representation of the distribution of the chosen metrics across the smart contracts. The boxplots reveal the central tendency, spread, and outliers for each metric, providing insights into the complexity and design patterns of the contracts under examination.

- Inheritance Depth: the majority of contracts exhibit low inheritance depths, indicating a preference for simpler inheritance structures. The presence of outliers suggests a few contracts employ deeper inheritance chains, potentially indicating more complex interactions or feature integrations.
- Coupling Between Contracts (CBC): similar to Inheritance Depth, the Coupling Between Contracts metric predominantly shows low values, implying limited inter-contract dependencies. This could reflect a design choice aimed at minimising coupling to enhance modularity and reduce complexity.
- State Variable Count: this metric displays a wider range of values with a noticeable spread, indicating variability in how contracts manage state. Contracts with a higher number of state variables may be handling more data or more complex state management tasks.
- Average Local Variables: most contracts tend to use a lower average number of local variables in their functions, suggesting a tendency towards functions with simpler logic. Outliers in this metric highlight contracts with functions that might be managing more complex calculations or data manipulations.
- Maximum Local Variables: exhibiting significant variability, this metric points to the complexity within the most complex function of each contract. Contracts with higher values may contain functions of considerable complexity, potentially impacting readability and maintainability.
- Number of Functions: the distribution of the number of functions per contract varies, with a trend towards a smaller number of functions. However, outliers indicate some contracts with a large number of functions, which could suggest a broader scope of responsibilities or functionalities encapsulated within these contracts.

Overall, the boxplots show the diversity in structural complexity and design choices across the smart contracts analysed. They highlight the balance between functionality and complexity in smart contract development, with outliers indicating instances of contracts that deviate from common patterns, either through increased complexity or through a significantly different approach to contract design.

Figure 5 offers a visualisation of function-level metrics across unique functions extracted from the "0aFullReport.csv" file. These metrics include the Number of Parameters, Nesting Depth, Function Calls, Cyclomatic Complexity, and Local Variable Count. Each boxplot encapsulates the distribution of its respective metric, highlighting median values, quartile ranges, and the presence of outliers, thereby illustrating the variability and complexity within the functions of smart contracts.

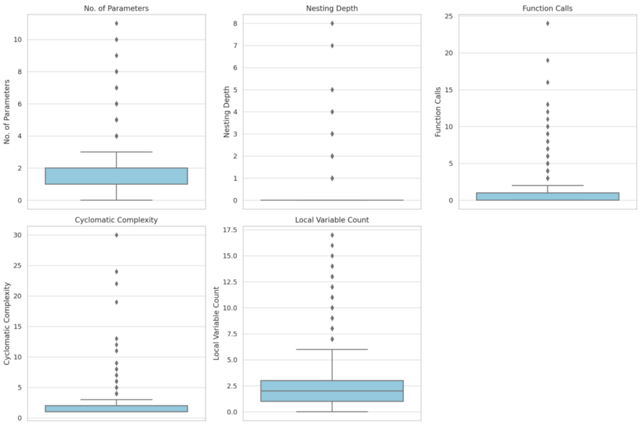

Figure 5: Distribution of Function Metrics

- Number of Parameters: the distribution for this metric indicates that most functions are designed with a limited number of parameters, suggesting a preference for simpler interfaces.

Outliers hint at functions that require more inputs, potentially reflecting more complex operations or interactions.

- Nesting Depth: this metric shows a general tendency towards lower nesting levels within functions, promoting readability and maintainability. However, functions represented as outliers demonstrate deeper nesting, which could indicate more complex logical structures or conditionals.
- Function Calls: the plot reveals a wide range of values, with a significant number of functions making few to no external calls. Outliers in this metric suggest functions with extensive interactions with other functions or contracts, potentially increasing the interdependence and complexity within the contract's ecosystem.
- Cyclomatic Complexity: the cyclomatic complexity measures the number of linearly independent paths through a program's source code. The distribution suggests that many functions maintain a lower complexity level, facilitating testing and maintenance. Functions marked as outliers exhibit higher complexity, indicating more intricate control flow which could impact understandability and error-proneness.
- Local Variable Count: most functions use a modest number of local variables, which aids in keeping the function's logic straightforward. Outliers, however, indicate functions with a high local variable count, pointing towards more complex computations or state management within those functions.

In our qualitative analysis, we chose to classify the function metrics in relation to the potential impact of the vulnerabilities (column Impact of the csv). The boxplots below provide an illustrative comparison across the varying levels of impact ('–', Low, Informational, Medium, High, Optimization) for three different metrics: Function calls, Cyclomatic Complexity, and Number of Parameters at function level, the first boxplot (related to '–') indicates functions in the dataset found to be without vulnerabilities.

In the first plot, Fig. 6, analysing the Function Calls, we observe a higher median and a wider interquartile range in contracts marked with "High" impact, suggesting that functions with a critical impact on contract behaviour tend to engage more frequently with other functions. The "Optimization" category shows a lower median, implying that functions within this category are likely more self-contained, possibly indicating a design that's optimised for gas efficiency or other performance measures.

The Cyclomatic Complexity plot (Fig. 7) illustrates a noticeably higher median in the "High" impact category as well, which could indicate a correlation between the complexity of a function's logic and the potential impact of its vulnerabilities. It's evident that as the perceived impact rises, so does the complexity, underlining the increased risk and the need for review and testing.

The plot for the Number of Parameters (Fig. 8) shows a generally increasing trend from "Low" to "High" impact, hinting that functions deemed to have a more significant impact may have more complex interfaces, taking in multiple parameters. This could be reflective of the functions' multifaceted nature and the critical role they play in the contract's operations.

Overall, comparing across categories, 'High' impact functions tend to exhibit higher complexity and interaction, as evidenced by greater median values and spreads in the related metrics. 'Optimization' functions often show the opposite, with reduced complexity

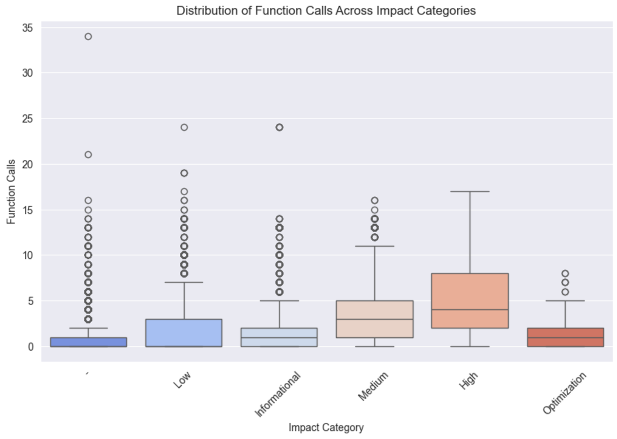

Figure 6: Distribution of Function Calls

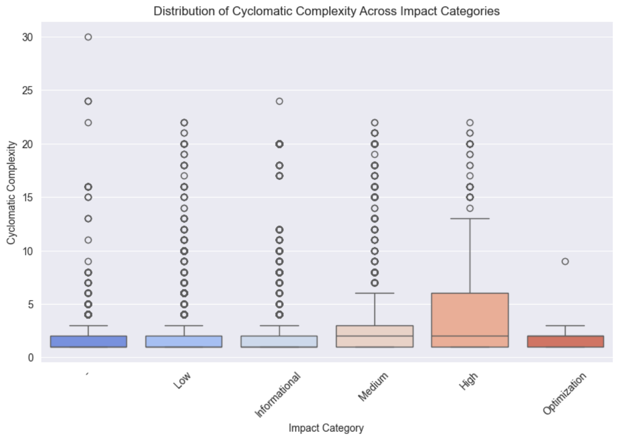

Figure 7: Distribution of Cyclomatic Complexity

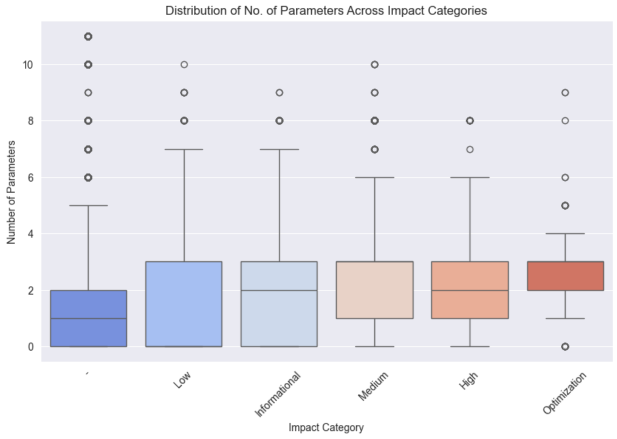

Figure 8: Distribution of Number of Parameters

and interactions, suggesting a focus on streamlined efficiency. This differentiation in metrics distributions across impact categories highlights the potential need for varied approaches to testing, review, and optimisation in smart contract development.

6 RESEARCH OPPORTUNITIES

The dataset presented in this study opens several avenues for future research, particularly in the field of predictive models and data analytics. Here, we outline potential directions for using this dataset:

- Predictive modeling for vulnerability detection: the set of vulnerability data and software metrics in our dataset provides an ideal basis for developing predictive models aimed at identifying potential security vulnerabilities in smart contracts. Researchers can use machine learning algorithms to analyse patterns and correlations between metrics and known vulnerabilities, creating models that can predict the likelihood of vulnerabilities in unseen contracts.
- Enhancing security practices through data analytics: the dataset can be used to conduct data-driven analysis on common coding practices and their impact on contract security. This could lead to the development of guidelines and best practices for writing safer smart contracts, supported by empirical evidence from the dataset.
- Automated tools for smart contract analysis: there is a significant opportunity to develop automated tools that integrate predictive models derived from the dataset. Such tools could assist developers in real-time by providing warnings or suggestions for improving contract security during the development process.
- Evaluating the effectiveness of security measures: by comparing contracts with known vulnerabilities against those that have employed certain security measures, researchers can quantitatively assess the effectiveness of different security practices and tools in mitigating vulnerabilities.

7 THREATS TO VALIDITY

Dataset representativeness: our dataset is drawn exclusively from the Smart Sanctuary dataset, containing over 200,000 verified smart contracts from Ethereum's main network. This collection, while extensive, may not capture every contract coding practice. Such omissions could affect our understanding of the relationships between software metrics and the occurrence of vulnerabilities. Additionally, our focus on main network contracts means we might miss out on valuable insights from contracts on test networks. These environments often serve as experimental grounds where developers test new ideas, which could provide unique case studies that enhance our dataset and subsequent analyses.

However, including contracts from test networks introduces its own set of challenges. Test networks are playgrounds for developers, leading to the deployment of multiple iterations of the same contract for testing purposes. This could significantly increase the presence of duplicate contracts in our dataset, complicating the analysis. Distinguishing between genuine variations and mere duplicates would require a more detailed approach to dataset curation, ensuring that our repository remains both complete and relevant to our research objectives. In future work, developing strategies to incorporate test network contracts without compromising data quality will be an important step towards a more representative and insightful dataset.

Contracts compilation issues: successfully compiling smart contracts from external repositories presents a set of challenges. A key issue involves selecting the correct compiler version, especially for contracts with pragma directives that span multiple versions. This task is far from straightforward, as it requires a detailed analysis of contract functionalities to determine the most suitable compiler version for accurate compilation.

Handling external dependencies adds another layer of complexity. Smart contracts frequently do not include a complete list of their dependencies, complicating the compilation process. While certain dependencies are readily available through package managers like *npm*—for instance, *@openzeppelin/contracts*[8]—others may be unique or bespoke to the contract, lacking mainstream package management support. Ensuring that these dependencies not only are present but also are compatible with the contract's specified pragma version adds to the challenge.

The situation is further complicated when contracts are developed using specific frameworks like *Truffle*[9] or *Hardhat*[10]. In such cases, key dependencies tied to these development environments might not be explicitly listed, hindering straightforward compilation. Addressing these complications demands an approach, involving both the tracking of dependencies and a flexible strategy for compiler version selection. Future efforts in this area might focus on developing tools or methodologies that streamline the compilation process, especially for contracts with complex dependency requirements or those developed within popular blockchain development frameworks.

Vulnerability detection tools limitations: in our study, we employ Slither for vulnerability detection, acknowledging its status as a leading tool in the field. Despite Slither's effectiveness and widespread use, it is important to recognise its inherent limitations, particularly its reliance on static analysis. This methodology prevents Slither from identifying vulnerabilities that require dynamic analysis to uncover, such as certain types of denial of service (DoS) issues. Slither shows constrained capabilities in spotting specific vulnerabilities like arithmetic overflows and underflows [13], indicating areas where its analysis might be supplemented.

To address these gaps, incorporating additional vulnerability detection tools could significantly enrich our dataset and refine our findings. Yet, the challenge lies in selecting the most appropriate tools to complement Slither. An exhaustive application of every available open-source vulnerability detection tool is unrealistic within the bounds of time and resource constraints. A more practical approach would involve a carefully chosen subset of tools, including dynamic analysis solutions like Mythril, to capture a broader spectrum of vulnerabilities not detectable through static analysis alone.

It is also important to consider tool compatibility with contract pragma versions. Tools such as Osiris and Oyente have limitations,

[8] https://www.npmjs.com/package/@openzeppelin/contracts
[9] https://archive.trufflesuite.com/
[10] https://hardhat.org/

mainly supporting contracts written in Solidity version 0.4. Therefore, tool selection must take into account not just the breadth of vulnerabilities covered but also compatibility with the range of pragma versions present in our dataset. Future enhancements to our methodology could include a strategic integration of both static and dynamic analysis tools, calibrated to effectively cover a comprehensive range of vulnerabilities across various Solidity versions.

8 FUTURE WORKS

Recognising the limitations identified in our analysis, our future efforts will focus on several key improvements. We aim to continuously update and expand our dataset by incorporating newer contracts from the Smart Sanctuary and other repositories, thereby enhancing the diversity and volume of vulnerability reports and software metrics available for analysis. A critical aspect of our forthcoming work involves refining the selection and configuration of vulnerability detection tools. We see significant value in integrating Mythril into our toolkit. Its dynamic analysis capabilities are expected to complement Slither's static approach, allowing us to uncover a broader array of vulnerability patterns. Furthermore, we plan to tackle the challenges of smart contract compilation and dependency resolution head-on. The lack of comprehensive tools and frameworks in current academic literature to address these specific issues underscores a pressing need. As a response, we are committed to developing a novel framework or toolset designed to streamline the compilation process and manage dependencies more effectively, addressing a critical gap in smart contract development and analysis.

9 CONCLUSIONS

This work has introduced a repository of Ethereum smart contracts, compiled to serve as a foundational tool for advancing research in smart contract security. By integrating detailed vulnerability reports generated by Slither with a wide set of software metrics at both contract and function level, we have provided researchers and developers with a rich resource for exploring the complex interplay between contract characteristics and security vulnerabilities. Our analysis, rooted in the examination of 50k contracts from the Smart Sanctuary dataset, has revealed insights into prevalent vulnerabilities and coding patterns, emphasising the need for ongoing vigilance and innovation in smart contract development practices.

The challenges identified in compiling and analysing smart contracts—ranging from the selection of appropriate compiler versions to the handling of external dependencies—underscore the complexities of ensuring contract security in a evolving ecosystem. Moreover, the reliance on static analysis tools like Slither, has highlighted the limitations of current methodologies in capturing the full spectrum of potential vulnerabilities.

Looking ahead, we are committed to expanding our dataset and refining our analysis methodology. The integration of dynamic analysis tools, along with the development of frameworks designed to streamline the compilation process, represents key avenues for future work. These efforts will enhance the robustness of our dataset and contribute to a deeper understanding of smart contract vulnerabilities.

Our dataset is available on Zenodo[11] offering a repository encompassing valuable resources such as smart contracts source codes, and associated software metrics and vulnerability reports, encouraging researchers and developers to enhance the current literature in SC security and analysis.

REFERENCES

[1] Giuseppe Antonio Pierro and Roberto Tonelli. 2020. PASO: A Web-Based Parser for Solidity Language Analysis. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. 16–21. https://doi.org/10.1109/IWBOSE50093.2020.9050263

[2] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 454–469.

[3] Monika di Angelo, Thomas Durieux, João F. Ferreira, and Gernot Salzer. 2023. Evolution of Automated Weakness Detection in Ethereum Bytecode: a Comprehensive Study. *Empirical Software Engineering* (2023). to appear.

[4] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*. 530–541.

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

[6] Giacomo Ibba, Sabrina Aufiero, Silvia Bartolucci, Rumyana Neykova, Marco Ortu, Roberto Tonelli, and Giuseppe Destefanis. 2024. Mindthedapp: a toolchain for complex network-driven structural analysis of ethereum-based decentralised applications. *IEEE Access* (2024).

[7] Giacomo Ibba, Giuseppe Destefanis, Rumyana Neykova, Marco Ortu, Sabrina Aufiero, and Silvia Bartolucci. 2024. DAI: A Dependencies Analyzer and Installer For Solidity Smart Contracts. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.

[8] G. Ibba, S. Khullar, E. Tesfai, R. Neykova, S. Aufiero, M. Ortu, S. Bartolucci, and G. Destefanis. 2024. A Preliminary Analysis of Software Metrics in Decentralised Applications. In *Proceedings of the Fifth ACM International Workshop on Blockchain-Enabled Networked Sensor Systems* (, Istanbul, Turkiye,) *(BlockSys '23)*. Association for Computing Machinery, New York, NY, USA, 27–33. https://doi.org/10.1145/3628354.3629533

[9] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 254–269.

[10] 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)*. IEEE, 1186–1189.

[11] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 778–788.

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

[13] Jarno Ottati, Giacomo Ibba, and Henrique Rocha. 2023. Comparing smart contract vulnerability detection tools. (2023).

[14] Giuseppe Antonio Pierro, Roberto Tonelli, and Michele Marchesi. 2020. An organized repository of ethereum smart contracts' source codes and metrics. *Future internet* 12, 11 (2020), 197.

[15] 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*. 9–16.

[16] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th annual computer security applications conference*. 664–676.

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

Received 2024-03-28; accepted 2024-04-19

[11] https://zenodo.org/records/11075555