HÁSKÓLINN Í REYKJAVÍK
REYKJAVIK UNIVERSITY

Advancing Smart Contract Security: Vulnerability Characterization, Classification, and Automated Detection

by
Majd Radwan Mohammad Soud

Dissertation submitted to the School of Technology,
Department of Computer Science
at Reykjavík University in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
October 9, 2024

Thesis Committee:

Grischa Liebel, Supervisor,
Associate Professor, Reykjavík University, Iceland.

Mohammad Hamadqa, Co-Supervisor,
Assistant Professor, Reykjavík University, Iceland
Assistant Professor, Polytechnique Montréal, Canada.

Alexander Norta,
Senior Researcher, Tallinn University, Estonia.

Yngvi Björnsson,
Professor, Reykjavík University, Iceland.

Examiner:

Manar Alalfi,
Associate Professor, Toronto Metropolitan University, Canada.

Advancing Smart Contract Security: Vulnerability Characterization, Classification, and Automated Detection

Majd Radwan Mohammad Soud

October 9, 2024

Abstract

Context: Smart contracts are computer programs deployed on the blockchain with significant value of cryptocurrency. They automate transactions and asset transfers, eliminating the need for intermediaries. These contracts serve as the foundation of decentralized applications (DApps), driving blockchain growth. However, their value and role make them attractive targets for attackers, leading to financial losses estimated at around $6.45 billion.
Objectives: This PhD dissertation aims to improve the security of smart contracts by systematically identifying, characterizing, and automatically detecting vulnerabilities in their code, hence advancing a more secure blockchain environment.
Method: We employ repository mining, qualitative analysis, and data science techniques. Specifically, we utilize repository mining to extract Ethereum smart contract vulnerabilities from public coding platforms and vulnerability databases. Moreover, qualitative methods such as open card sorting are employed to characterize the extracted vulnerabilities. Finally, data science techniques, including machine learning algorithms, are applied to automatically detect and prioritize vulnerabilities early in the smart contract lifecycle.
Results: Our research demonstrates the effectiveness of the mentioned methods in identifying, characterizing, and detecting vulnerabilities in smart contracts. Our findings reveal significant novel vulnerabilities in Ethereum smart contracts from the selected repositories. Through qualitative and quantitative analysis, we provide insights into the distribution of these vulnerabilities across several data sources, their characteristics, and dimensions, including error sources and impacts. Furthermore, this PhD dissertation provides a unified and comprehensive taxonomy of smart contract vulnerabilities. We offer a framework and a suite of tools for automated vulnerability mining, classification, prioritization, and detection using data science techniques to improve the overall security of smart contracts. Finally, several mitigation strategies, quantitatively extracted

from real-world smart contract code changes, are presented, along with recommendations and implications for researchers and practitioners.
Conclusions: In conclusion, this PhD dissertation highlights the importance of securing Ethereum smart contracts by focusing on vulnerability key characteristics, automated prioritization, and detection to prevent cyberattacks. Despite encountering challenges, such as the need for unified data and extensive resources, this PhD dissertation offers valuable insights into smart contract security during the development process. Another key challenge was addressing logic vulnerabilities, which required a more advanced understanding of code semantics and proved particularly complex compared to syntactic vulnerabilities. Future work should focus on investing in advanced semantic analysis to effectively mitigate complex vulnerabilities. Our findings enable researchers, practitioners, and tool builders to better understand smart contract vulnerabilities and strengthen security policies and tools using our datasets, tools, and framework.

Framfarir í snjallsamningaöryggi: Veikleikagreining, flokkun og sjáfvirk uppgötvun.

Majd Soud

October 9, 2024

Útdráttur

Snjallsamningar eru forrit á bálkakeðjum sem varða mikil verðmæti af rafmynt. Þau sjálfvirknivæða færslur og millifærslu á eignum, og afnema Þar með Þörfina á milliliðum. Slíkir samningar Þjóna sem grunnur fyrir dreifð forrit (DApps), og eru Þannig drifkraftur í vexi bálkakeðja. Hins vegar, gera verðmæti Þeirra og hlutverk Þá að verðugu takmarki fyrir árásaraðila, sem leiðir til fjárhagslegs taps sem metið er að 6,45 milljörðum dollara.
Þessi doktorsritgerð miðar að Því að auka öryggi snjallsamninga með Því að greina, einkenna og greina kerfisbundið veikleika í kóða Þeirra og Þar með Þróa öruggara bálkakeðjuumhverfi.
Við notum gagnanám, eigindlega greiningu og gagnavísindalegar aðferðir. Nánar tiltekið notum við gagnanám til að finna veikleika í Ethereum snjallsamningum úr opinberum kóðunarverkvöngum og veikleikagagnagrunnum. Auk Þess eru eigindlegar aðferðir eins og opin kortaflokkun notaðar til að lýsa veikleikunum sem finnast. Að lokum er gagnavísindalegum aðferðum, Þar með talið vélanámi, beitt til að greina og forgangsraða veikleikum snemma í lífsferli snjallsamninga. Rannsóknir okkar sýnir fram á árangur áðurnefndra aðferða við að greina, einkenna og finna veikleika í snjallsamningum. Niðurstöður okkar leiða í ljós umtalsverða nýbreytni í veikleikum í Ethereum snjallsamningum. Með eigindlegri greiningu fáum við innsýn í dreifingu veikleikanna á milli nokkurra gagnasafna, einkenni Þeirra og víddir, Þ.m.t. skekkjuvalda og áhrif. Auk Þess veitir Þessi doktorsritgerð sameinaða og yfirgripsmikla flokkun á veikleikum í snjallsamningum. Við bjóðum upp á umgjörð og tól til sjálfvirkrar vörpunar veikleika, flokkunar, forgangsröðunar og greiningar með gagnavísindalegum aðferðum til að bæta heildaröryggi snjallsamninga. Að lokum eru kynntar nokkrar aðferðir til að draga úr veikleikum, byggðar á raunverulegum breytingum á kóða snjallsamninga, ásamt tillögum og afleiðingum fyrir rannsakendur og fræðimenn.
Í Þessari doktorsritgerð er lögð áhersla á mikilvægi Þess að tryggja öryggi Ethereum snjallsamninga með Því að beina sjónum að lykileiginleikum veikleika,

sjálfvirkri forgangsröðun og greiningu til að koma í veg fyrir netárásir. Þrátt fyrir að viðfangsefnin séu mörg, svo sem Þörfin fyrir sameinuð gögn og umfangsmikilar auðlind, Þá veitir doktorsverkefnið mikilvæga innsýn í snjallsamningavernd í Þróunarferlinu. önnur lykilviðfangsefni voru röklegir veikleikar, sem kröfðust dýpri skilnings á merkingarfræði og reyndust sérlega flókin í samanburði við málskipanar veikleika. Næstu skref í Þessum rannsóknum er að leggja áherslu á að fjárfesta í Þróaðri merkingarfræðilegri greiningu til að draga úr flóknum veikleikum á skilvirkan hátt. Niðurstöður okkar gera rannsakendum, fagaðilum og hugbúnaðarsérfræðingum kleift að skilja betur veikleika snjallsamninga og styrkja öryggisstefnur og verkfæri með Því að nota gagnasöfnin okkar, verkfæri og umgjörð.

To my beloved husband,
my steadfast companion,
the love of my life,
the one who faced the world's uncertainties to support my dreams.
For all the times I could only express my gratitude in whispers,
today I proudly dedicate this thesis to you.

To Yamin,
who enveloped me in warm hugs and bright smiles,
your love and joy gave me the strength to continue this journey.

Acknowledgments

Four years of my PhD journey, excluding a break for maternity leave, have been a roller-coaster ride filled with many ups and downs. This journey has significantly enhanced my research skills and personal growth. More than just an achievement, it has been made possible by the incredible support of those around me.

First, I would like to express my gratitude to my supervisor, Dr. Grischa Liebel, for granting me the opportunity and freedom to explore my research interests. His reliable support and invaluable counsel were always there when I needed them. His enduring encouragement and confidence in my abilities are key reasons I have reached this point in my journey.

I would like to extend my thanks to my co-advisor, Dr. Mohammad Hamadqa, for recruiting me into the Reykjavík University PhD research program, CRESS group, and the Fintech group. This opportunity allowed me to gain valuable experience in managing meetings, overseeing student projects, and presenting research papers. I am grateful for the foundational opportunity that significantly contributed to building my academic journey.

I wish to express my sincere gratitude to the members of my PhD defense committee, including Dr. Alexander Norta, Dr. Manar Alalfi, and Dr. Yngvi Björnsson. It is a great honor that they have agreed to serve on my committee, and I deeply value the opportunity to have my thesis reviewed by such esteemed members of the scientific community.

I also wish to acknowledge the financial support provided by the Icelandic Research Fund (Rannís). This funding played a critical role in facilitating my research and the successful completion of this PhD.

This PhD journey also provided me the chance to work with remarkable and distinguished professors at the Fintech Group and CRESS Group. It was my pleasure to collaborate with Dr. Gísli Hjálmtýsson on several research papers and projects. I had an interesting time discussing research in the field with him,

and his insights and expertise have been invaluable.

I would also like to give special thanks to Prof. Luca Aceto for his insightful advice, support during PhD meetings, and overall kindness. Additionally, I am thankful to Prof. Anna Ingólfsdóttir for her continuous support and encouragement throughout this journey.

I am grateful to have had the opportunity to meet and work with professors including Björn Þór Jónsson, María óskarsdóttir, Anna Sigríður Islind, Hans Peter Reiser, Stefán ólafsson, Gylfi Þór Guðmundsson, and Marta Kristín Lárusdóttir. Their invaluable feedback during numerous meetings at both the CRESS group and the Fintech group has been greatly appreciated.

Last but certainly not least, I want to express my deepest gratitude to my husband, Mohammad. Without his steadfast support, this thesis would not only have been incomplete, but my life would have felt profoundly different. Mohammad has stood by me through every step of this journey, offering love, encouragement, and sacrifice during the challenging times, and sharing in the joy of the good times. Thank you for being my rock, enduring the ups and downs, taking care of me, and inspiring me to strive for more. Your patience and understanding have meant the world to me, and your presence has been a source of strength and motivation. Without you, this accomplishment would not have been possible. I am deeply grateful, my love!

I want to extend my heartfelt thanks to those outside the scientific community who supported me throughout this journey. Whether it was my mother, father, sister, family, or friends, their companionship and care provided the strength I needed to persevere and made this experience truly meaningful.

Finally, I am grateful to the kindergarten system here in Iceland, which pushed me to complete my thesis writing before the summer break of 2024 so I could spend quality time bonding with my child, Yamin. I would like to extend my heartfelt thanks to everyone who took care of Yamin while I was busy working on my thesis. Your support has been invaluable and allowed me to focus on my research with peace of mind.

List of Publications

Appended publications

This thesis is based on the following publications:

1. **Soud, M., Liebel, G., & Hamdaqa, M. (2024).** A fly in the ointment: an empirical study on the characteristics of Ethereum smart contract code weaknesses. *Empirical Software Engineering*, 29(1), 13. (Accepted 2023)
Contribution: I proposed the research idea, collected and manually preprocessed the data, led the development of the methodology and results, and wrote the majority of the manuscript.

2. **Soud, M., Qasse, I., Liebel, G., & Hamdaqa, M. (2023, September).** Automesc: Automatic framework for mining and classifying Ethereum smart contract vulnerabilities and their fixes. In *2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (pp. 410-417). IEEE.
Contribution: I proposed the research idea, identified the research gap, implemented the framework, collected and curated the dataset, and wrote the majority of the manuscript except for the evaluation section. I also presented the paper at the SEAA conference.

3. **Soud, M., Liebel, G., & Hamdaqa, M. (2023).** PrAIoritize: Automated early prediction and prioritization of vulnerabilities in smart contracts. *arXiv preprint* arXiv:2308.11082. *In submission to a journal.*
Contribution: I proposed the research idea, implemented the automated prediction, collected and prepared the data for analysis, and wrote the majority of the manuscript.

4. **Soud, M., Nuutinen, W., & Liebel, G. (2024).** Sóley: Identification and automated detection of logic vulnerabilities in Ethereum smart contracts using large language models. *arXiv preprint* arXiv:2406.16244. *Submitted to Journal of Systems and Software.*
Contribution: I proposed the research idea, supervised a master's degree student, participated in the development of the logic vulnerability detection framework using large language models, collected the data, and wrote the majority of the manuscript.

List of Tables

List of Figures

Contents

Chapter 1

Introduction

Following the success of Bitcoin, a decentralized cryptocurrency that reached a market capitalization of $1.35 trillion in 2024 [1], national governments and industries are captivated by the potential of blockchain [33]. A blockchain is an append-only distributed data structure managed by the nodes of a peer-to-peer network [30]. Cryptocurrencies utilize the blockchain as a public ledger to record all currency transfers, thereby preventing double-spending [4]. While Bitcoin exemplifies blockchain technology, blockchain technology has applications far beyond cryptocurrencies, including financial services, property tracking, digital identity verification, and voting [50]. Ethereum [6], a pioneering platform that extended blockchain technology to facilitate such applications, achieved a capitalization of $468.1 billion in 2024 [2]. Ethereum gained prominence for being the first blockchain to introduce smart contracts. The term *"smart contract"* is commonly employed in two distinct contexts [22]. First, operationally, it involves software agents on a shared ledger executing obligations and controlling assets. Second, it refers to expressing and implementing legal contracts in software, addressing contract drafting and legal interpretation nuances [22].

In this dissertation, the term "smart contract" adheres to the following definition [10]:

> ***"A smart contract is an automatable and enforceable agreement. Automatable by computer, although some parts may require human input and control. Enforceable***

[1] https://coinmarketcap.com/currencies/bitcoin/
[2] https://coinmarketcap.com/currencies/ethereum/

either by legal enforcement of rights and obligations or via tamper-proof execution of computer code."

In Ethereum, smart contracts are typically written using the Solidity high-level programming language and then compiled into bytecodes for execution by the Ethereum Virtual Machine (EVM) [53]. As each underlying technology imposes its unique characteristics on applications running atop it, blockchain also introduces critical aspects affecting deployed smart contracts. Inheriting immutable, self-executing, and decentralized attributes from the underlying blockchain technology, smart contracts are extremely challenging to modify once deployed to the blockchain, as their execution relies entirely on their unchangeable code [53]. These characteristics ensure the reliability of smart contracts and distinguish them from conventional software, but they also pose significant development challenges.

The first challenge is ensuring error-free implementation pre-deployment. Given the immutability of smart contracts once deployed, vulnerabilities in the code of smart contracts can have far-reaching consequences. A notable example is the Governmental contract [6], where $2.5 million worth of Ether [3] got locked out due to a vulnerability in the smart contract code. Another example is the security breach involving the Poly Network in 2021 [1], where hackers exploited a vulnerability in a smart contract to steal over $600 million worth of cryptocurrency.

These incidents further highlight the critical importance of identifying and addressing vulnerabilities in smart contracts before deployment.

The second challenge is the cost of execution, measured in gas on Ethereum. Unoptimized code can lead to excessive gas consumption and financial losses, especially for decentralized applications that cannot sustain high transaction costs [20]. Ethereum's "block gas limit" further complicates matters, as exceeding this limit causes transaction failures and can lock users out of their funds.

Third, while the transparency of blockchain is crucial for trust, it raises confidentiality and security concerns, particularly when handling sensitive information. For example, vulnerabilities such as timestamp manipulation or front-running attacks complicate security measures and have led to substantial financial losses [55].

Ensuring the proper execution of Ethereum smart contracts is essential, but it does not guarantee security. Numerous vulnerabilities in smart contract code have led to significant financial losses from cyberattacks. Therefore, enhancing smart contract vulnerability management is crucial to safeguarding digital as-

[3] Ethereum cryptocurrency

sets, ensuring transaction integrity, and improving overall blockchain security.

In this dissertation, we consider vulnerabilities in smart contract code. According to the National Vulnerability Database (NVD) [46], a vulnerability is defined as:

> *"A weakness in the computational logic (e.g., code) found in software and hardware components that, when exploited, leads to adverse effects on confidentiality, integrity, or availability. Addressing such vulnerabilities usually requires modifications to the codebase, but may also entail alterations to specifications or even the deprecation of certain specifications (e.g., eliminating affected protocols or functionalities entirely)".*

This definition aligns with the ISO/IEC 23643 standard, which characterizes vulnerabilities as potential flaws or weaknesses in software design or implementation. These flaws can be unintentionally triggered or deliberately exploited, resulting in harm to the system [35].

Smart contract vulnerabilities are diverse and complex, often resulting from a misalignment between Solidity's semantics and programmers' intuition [4]. Additionally, Solidity lacks constructs to handle domain-specific aspects, such as recording computation steps on a public blockchain where they can be unpredictably reordered or delayed. The scattered documentation across official sources [5, 2], research papers [11, 54, 13], and forums [4] adds to the complexity. Despite the available information, a comprehensive and up-to-date characterization and standardization of Ethereum smart contract vulnerabilities and security measures is still missing. Ensuring the security of smart contracts is essential in software engineering, requiring an in-depth analysis of vulnerabilities and accurate detection during the development of smart contracts.

PhD Dissertation Aim

The main aim of this dissertation is to improve the security of Ethereum smart contracts by systematically identifying, characterizing, and detecting vulnerabilities in their code.

Toward this end, we break down our main aim into the following goals:

- **G1**: Identify, characterize, and categorize vulnerabilities in Ethereum smart contracts to understand their types, sources of errors, and impacts.

[4] https://ethereum.stackexchange.com/

- **G2**: Improve the automated detection and classification techniques to detect vulnerabilities and enhance the overall security of Ethereum smart contracts.

Our research employs repository mining, qualitative analysis, and data science techniques. Specifically, we leverage repository mining to define novel vulnerabilities and systematically examine their distribution across diverse data sources. Additionally, we use qualitative analysis techniques, such as card sorting, to pinpoint the sources and impacts of vulnerability errors.

Furthermore, we apply data science techniques, including machine learning algorithms, to develop automated tools and frameworks for detecting and prioritizing these vulnerabilities. By adopting repository mining, qualitative analysis, and data science techniques, our research advances smart contract security.

Despite the potential of these methods, our research faced challenges in synthesizing diverse data formats and ensuring data quality from various sources. Moreover, analyzing logical vulnerabilities proved particularly complex, requiring advanced semantic understanding compared to syntactic vulnerabilities.

The remainder of this introduction chapter is organized as follows. In Section 2.2, we lay the foundation by providing background information about smart contracts and the Ethereum blockchain. Section 1.2 discusses the goals and scope of this dissertation in more detail. Related work is outlined in Section 1.3, followed by the research methodology in Section 1.4. The contribution of each individual paper is discussed in Section 1.5. Subsequently, the findings are summarized in Section 1.6, leading into a discussion of the overall dissertation topic in Section 1.7. Potential threats to the validity of this dissertation are addressed in Section 1.8. Finally, the introduction chapter concludes with a summary and a discussion of future work.

1.1 Background

This section provides essential background information for this PhD dissertation, including an overview of blockchain technology, smart contracts, and Solidity programming language.

1.1.1 Blockchain and Ethereum

Blockchains are transparent, decentralized ledgers of transactions [3]. A blockchain consists of a set of blocks; each block contains a sequence of transactions that

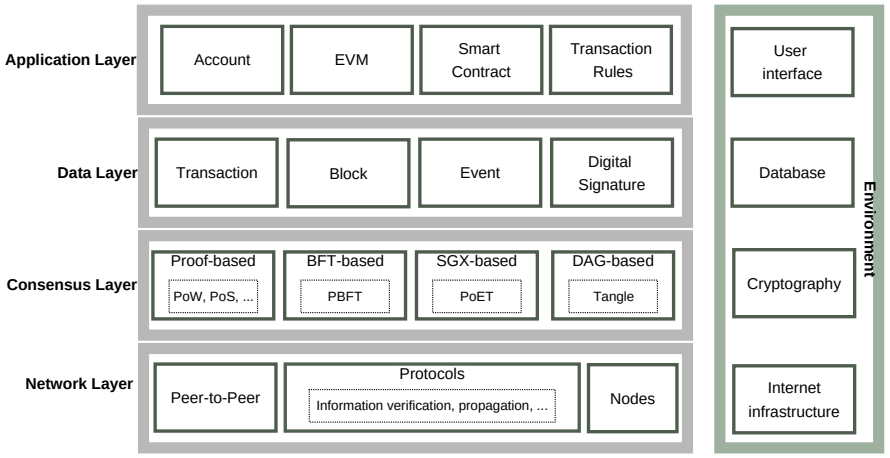


Figure 1.1: Ethereum blockchain architecture [47]

cannot be altered without affecting all subsequent blocks [57]. Miners maintain the blockchain by accepting the longest valid chain as the accepted version. To encourage transaction validation and prevent malicious actors, blockchain platforms such as Ethereum employ incentives and consensus mechanisms such as proof of work (PoW) and proof-of-stack (PoS) [53]. Instead of relying on a central authority to coordinate multiparty interactions, the majority consensus among miners determines the valid transactions in cases of conflict.

Figure 1.1 shows the 4-layer architecture of the Ethereum blockchain [47]. At its Network Layer, Ethereum operates as a peer-to-peer network where each node maintains a complete copy of the blockchain. The Consensus Layer employs a variant of consensus protocols to determine the main chain from competing valid blocks [53]. In the Data Layer, Ethereum manages transactions between externally owned accounts (EOAs) and contract accounts, which is crucial for updating the blockchain's state [18]. Finally, the Application Layer manages EOAs and contract accounts, facilitating interactions with decentralized applications (DApps) and smart contracts deployed on the Ethereum blockchain. Next, a detailed description of each layer is provided.

At the network layer, Ethereum operates as a structured peer-to-peer network where each node maintains a complete copy of the blockchain [24]. For

node discovery and routing, Ethereum nodes utilize a dynamic routing table with several buckets, each containing up to 16 entries of other nodes' identifiers, IP addresses, and communication ports [42]. The network relies on protocols such as RLPx for node discovery and the Ethereum Wire Protocol for efficient data exchange of blockchain information such as transactions and blocks among network participants[38]. This infrastructure ensures robust communication and synchronization across the decentralized network of Ethereum nodes [24].

At the consensus layer, Ethereum generates a new block approximately every 14 seconds, allowing multiple miners to concurrently produce valid blocks and occasionally resulting in stale blocks [59]. Ethereum employs a modified GHOST (Greedy Heaviest-Observed Sub-Tree) consensus protocol to select the main chain, favoring the sub-tree with the highest cumulative block difficulty [59]. While stale blocks are not included in the main chain, they enhance network security and receive rewards alongside regular blocks. Furthermore, uncle blocks, which are stale blocks referenced by regular blocks, incentivize network participation by providing additional rewards to both miners involved in the referencing process [30].

The data layer facilitates transactions between EOAs and contract accounts, incorporating essential components such as transaction nonce, recipient address, Ether amount, transaction data or bytecode, gas price, gas limit for transaction fees, and the sender's ECDSA [5] signature [53]. These transactions are pivotal in updating both account and blockchain states, progressing through a lifecycle that includes creation, validation, broadcasting, mining, and block validation. Each Ethereum contract account corresponds to a leaf or branch node on the state trie [6] [53] and manages persistent data using a dedicated storage trie structured with (key, value) pairs. This configuration links key positions directly to specific contract state variables stored in predefined slots. Unlike Bitcoin, Ethereum utilizes a single evolving state trie that dynamically updates to accurately reflect the current state of the blockchain [53].

In the application layer, EOAs hold user funds in Wei [7] and are secured by public-private key pairs. Contract accounts contain executable bytecode known as smart contracts, defining specific business logic with dynamic states tracked by nonce, balance, storageRoot, and codeHash. Ethereum's support for decentralized applications (DApps) spans sectors such as finance and governance,

[5] Elliptic Curve Digital Signature Algorithm, which is used in Ethereum to verify the authenticity of transactions and messages.

[6] A Merkle trie data structure in Ethereum that stores the current state of accounts.

[7] Wei is the smallest unit of Ether, where 1 Ether = 10^{18} Wei. For more information, see `https://academy.binance.com/en/glossary/wei`.

often involving the issuance of tokens such as ERC-20 tokens for Initial Coin Offerings (ICOs) and exchanges [11, 3].

Gas

In Ethereum, gas plays an important role in ensuring security, efficiency, and sustainability. Ethereum uses gas to measure the computational effort required to execute smart contracts. Each operation, such as storing data, performing calculations, or executing code, consumes a certain amount of gas [13]. Smart contracts and transactions must provide sufficient gas to cover their computational requirements as they interact with Ethereum network. When more gas is consumed than the sender provides, a transaction fails and the changes on the blockchain are reverted. This mechanism prevents infinite loops and other resource-intensive activities that could lead to denial of service (DoS) attacks. As a result of Ethereum's use of gas, users are incentivized to create cost-effective and sustainable smart contracts. Transaction fees are calculated using

$$GasPrice \cdot GasUsed \tag{1.1}$$

where gas refers to the amount of computing and storage resources consumed. By definition, GasPrice is constant, and GasUsed by a transaction is specified by Ethereum's core protocol [53].

1.1.2 Smart Contracts

The concept "**Smart Contracts**" was first coined by Szabo in 1994 [54], who defined them as "computerized transaction protocols that execute the terms of a contract". Smart contracts are self-executing digital agreements programmed with high-level languages that require deployment on the blockchain to execute. They possess three key characteristics: decentralization, immutability, and the ability to handle high financial values [13]. Decentralization ensures that no single entity controls them, promoting transparency and trust among diverse participants. Immutability means that it is extremely difficult and expensive to change their code once deployed on the blockchain, ensuring security and integrity. In addition, they facilitate financial transactions, automate payment processing, and enforce conditions for fund transfers, eliminating the need for intermediaries and reducing transaction costs [41]. These features make smart contracts valuable tools for creating secure, transparent, and efficient agreements within decentralized systems.

On Ethereum, smart contracts are typically written in high-level programming languages such as Solidity and Vyper [19]. These contracts enforce strict rules and consequences for involved parties under specified conditions, mirroring traditional legal contracts. Within their functions, smart contracts utilize specific parameters to execute transactions efficiently. For example, in a token transfer smart contract, parameters might include the sender's address, recipient's address, and the amount of tokens to transfer. During execution, these parameters control changes in the contract's variables based on its programmed logic. A smart contract deployed on Ethereum can be invoked by any user through a transaction containing the contract's address, signature, and input parameters for the targeted function. This transaction is sent to the nearest Ethereum node for processing. Smart contracts take various forms, such as token contracts defining token characteristics and overseeing their distribution among user accounts [19]. Additionally, these contracts can function independently as applications. For example, they often serve as the core or backend infrastructure for decentralized applications (DApps) [11].

Solidity and Vyper are high-level programming languages used to write smart contracts on the Ethereum blockchain. Each language serves different purposes and design principles. Solidity, known for its rich feature set, supports advanced programming constructs such as inheritance and libraries, offering high flexibility and widespread adoption with extensive tooling and documentation [8]. In contrast, Vyper emphasizes simplicity and security by deliberately excluding complex features such as inheritance to enhance code readability and minimize potential vulnerabilities [9]. While Vyper prioritizes minimalism and auditability, making contracts easier to understand and verify, it is less mature and has fewer development tools compared to Solidity.

A smart contract written in Solidity [10] consists of programmable functions and state variables, as illustrated in Figure 1.2. Transactions involving smart contracts require a specific set of parameters for functions within the same contract. During execution, the variables' states change according to the contract's logic implementation.

Solidity compiler translates the contract code into bytecode, which is executed by the EVM. This bytecode is similar to an assembly language, comprising low-level instructions called opcodes. Upon deployment, each smart contract on the blockchain network is assigned a unique address. Every node in the blockchain network executes the contract's bytecode as part of its process to

[8] https://soliditylang.org/
[9] https://docs.vyperlang.org/en/stable/
[10] https://solidity.readthedocs.io

```
+ pragma solidity >=0.7.0 <0.9.0; // compiler version

# contract Ballot { // contract declaration
*    struct Voter { // struct declaration
*        uint weight;  // uint weight
*        bool voted;   // bool voted
*        address delegate;  // address delegate
*        uint vote;    // uint vote
*    }

    struct Proposal {
        bytes32 name;   // bytes32 name
        uint voteCount; // uint voteCount
    }

    address public chairperson; // address chairperson
// mapping of address to Voter
    mapping(address => Voter) public voters;
    Proposal[] public proposals; // dynamic array of Proposal

-    constructor(bytes32[] memory proposalNames) { // constructor
        chairperson = msg.sender;
        voters[chairperson].weight = 1;
~        for (uint i = 0; i < proposalNames.length; i++) {
 // for loop
|            proposals.push(Proposal({ // push operation
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }

-    function winningProposal() public view returns
- (uint winningProposal_)
- { // function declaration
        uint winningVoteCount = 0;
// for loop
~        for (uint p = 0; p < proposals.length; p++) {
 // if statement
~            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }
}
```

Figure 1.2: A smart contract written in Solidity

verify and validate new blocks.

A smart contract can function as a token contract, which defines a token and manages its distribution among user accounts. Similarly, a contract can serve as a standalone application. For instance, it can operate as backend infrastructure or as a foundational component for complex applications, known as decentralized applications (DApps) [11]. Solidity provides various constructs to allow portioning concerns such as libraries and subcontracts. Libraries contain multiple functions designed to optimize processing time or handle specific scenarios such as a math library that prevents overflows and underflows. Subcontracts facilitate the creation of object-oriented relationships between contracts, similar to object-oriented programming (e.g., reflecting inheritance relationships between contracts).

To sum up, a contract comprises instructions, variables, methods, and programming components. Similar to traditional legal contracts, the code in smart contracts defines strict rules, commitments, benefits, and consequences applicable under specific conditions.

1.2 Scope and Goals

This PhD dissertation is organized around two goals (G), each addressed by specific research questions (RQs). These goals are explored across four chapters, with each chapter dedicated to a particular research question and its corresponding paper.

- **G1**: Identify, characterize, and categorize vulnerabilities in Ethereum smart contracts to understand their types, sources of errors, and impacts.
  - **RQ1:** How can vulnerabilities in Ethereum smart contracts be characterized using various data sources and unified with existing taxonomies based on multiple code vulnerability dimensions?
  - **RQ2:** To what extent can vulnerabilities in Ethereum smart contracts and their fixes be regularly and automatically mined, labeled, and categorized from various data sources to overcome limitations in existing datasets?
- **G2**: Improve the automated detection and classification techniques to detect vulnerabilities and enhance the overall security of Ethereum smart contracts.

- **RQ3:** To what extent can vulnerabilities in Ethereum smart contracts be early predicted and prioritized during code review processes?
- **RQ4:** How can historical code changes and advanced detection techniques be leveraged to effectively identify, detect, and mitigate logic-related vulnerabilities in smart contracts?

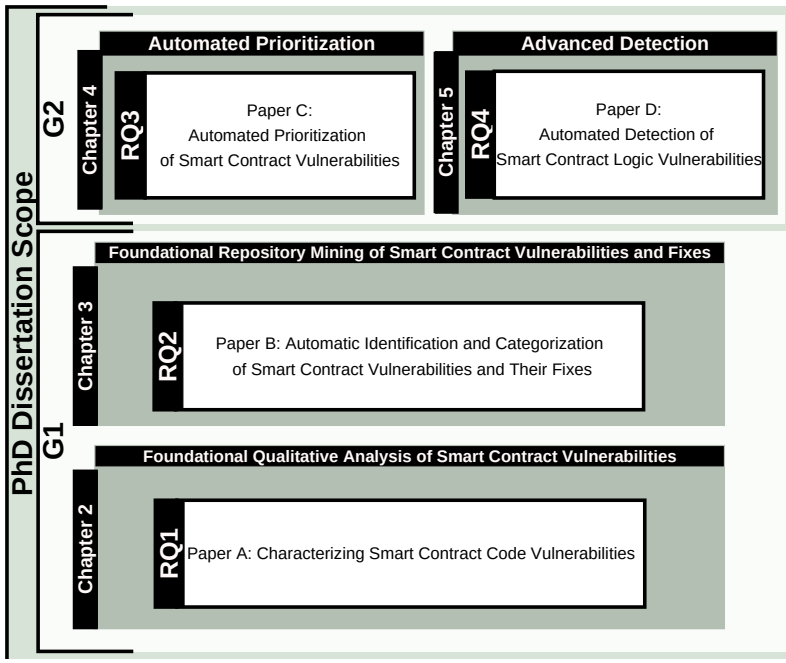


Figure 1.3: Scope overview of this PhD dissertation

The first goal (**G1**) aims to comprehensively characterize and categorize vulnerabilities in Ethereum smart contracts. **RQ1** aims to systematically characterize vulnerabilities from heterogeneous data sources and unify these findings with existing taxonomies based on several dimensions (e.g., source of error) using qualitative methods to establish a unified taxonomy for smart contract vulnerabilities. This foundational taxonomy and characteristics set the stage for **RQ2**, which complements **RQ1** by automating the identification, labeling,

and categorization of vulnerabilities and their fixes from diverse data sources through repository mining.

Therefore, the paper addressing **RQ2** categorizes these vulnerabilities based on the types outlined in the taxonomy developed in the paper addressing **RQ1** , thereby overcoming limitations in existing datasets and maintaining an up-to-date dataset of smart contract vulnerabilities.

Moreover, the second goal (**G2**) aims to enhance automated detection and classification techniques for detecting smart contract vulnerabilities and improving overall security. **RQ3** investigates the early prediction and prioritization of vulnerabilities during code review processes, crucial for timely addressing vulnerabilities identified under **G1**. **RQ3** utilizes data science techniques, the taxonomy from **RQ1**, and the dataset resulting from **RQ2** to automatically predict the priorities of smart contract vulnerabilities. **RQ4** explores the use of historical code changes and advanced detection techniques to effectively identify, detect, and mitigate logic-related vulnerabilities. **RQ4** contributes to the automated detection of vulnerabilities during smart contract development under **G2**, utilizing both data science techniques and qualitative analysis on the dataset resulting from **RQ2**.

Figure 1.3 illustrates the scope of this PhD dissertation through the contributions of Goals **G1** and **G2**, along with the related research questions (**RQ1**, **RQ2**, **RQ3**, and **RQ4**). Each research question is represented by its corresponding paper: Paper A, Paper B, Paper C, and Paper D, respectively.

Papers A and B are dedicated to achieving **G1** by laying the foundation for **G2** through identifying, characterizing, and categorizing vulnerabilities in Ethereum smart contracts. **Paper A** conducts a systematic analysis of vulnerabilities, synthesizing the resulting characteristics to establish a foundational taxonomy **(RQ1)**, which **Paper B** utilizes for the identification, labeling, and categorization of vulnerabilities and their fixes **(RQ2)**. **Paper B** contributes to maintaining a dataset of vulnerabilities and their fixes, essential for **G2**.

Paper C and Paper D contribute to **G2** by enhancing automated detection and classification techniques for smart contract vulnerabilities. **Paper C** addresses **RQ3** by automating the prioritization of vulnerabilities, utilizing characteristics from the taxonomy developed in **Paper A (RQ1)**. Finally, **Paper D** investigates advanced detection techniques **(RQ4)** to detect logic-related vulnerabilities in smart contracts. **Paper D (RQ4)** utilizes historical code changes extracted from the dataset established in **Paper B (RQ2)**.

1.3 Related Work

This section details the related body of knowledge relevant to this PhD dissertation. Considering the interconnected nature of the proposed papers, they share a substantial amount of related knowledge and literature. Therefore, the related work is organized by topics.

1.3.1 Categorization and Classification Schemes of Smart Contract Vulnerabilities

Since the first attack on smart contracts, numerous studies have emerged to classify smart contract vulnerabilities. These studies can be grouped into two main categories based on the data sources used. The first category relies on repository and forum discussions, as seen in the works of Chen et al. [11] and Zhang et al. [54]. Chen et al.'s study, for instance, utilized data collected from Ethereum StackExchange, where developer discussions were extracted and classified into five general categories: security, availability, performance, maintainability, and reusability. Their classification is solely based on the consequences of vulnerabilities. They also conducted an online survey to gather practitioners' opinions. However, a drawback of this classification is that the proposed categories often overlap and do not provide distinct differences, resulting in some vulnerabilities being classified into multiple categories. Another study by Zhang et al. [54] utilized two main data sources—open projects on GitHub and literature—to devise a classification scheme for smart contract vulnerabilities. The resulting classification contains nine categories, based on an extension of the IEEE Standard Classification for Software Anomalies[11]. Additionally, they proposed another classification scheme for the impact of a code vulnerability, which includes four categories: unwanted function executed, performance, security, and serviceability.

The second group consists of studies that depend exclusively on literature. Atzei et al. [4] classified smart contract vulnerabilities into three categories based on gray and white literature: language-related issues, blockchain issues, and EVM bytecode issues. On the other hand, Huang et al. [22] considered the software lifecycle perspective and discussed smart contract vulnerabilities across four development phases: design, implementation, testing before deployment, and monitoring and analysis. They classified ten vulnerabilities into three categories: Solidity, blockchain, and misunderstanding of common practices.

[11] https://standards.ieee.org/standard/1044-2009.html

Unfortunately, the study does not provide information on what basis these categories were designed. Furthermore, Samreen et al. [43] extended this body of knowledge by mapping eight identified smart contract vulnerabilities in the literature to the NIST-BF framework. Their findings indicated that among the eight vulnerabilities studied, only three could be associated with two specific classes within the NIST-BF framework. Similarly, Praitheeshan et al. [38] categorized 16 reported smart contract vulnerabilities into three distinct categories: blockchain-related issues, software security concerns, and challenges specific to Ethereum and Solidity. Lastly, Chen et al. [13] surveyed smart contract vulnerabilities, categorizing them into two dimensions: network location (e.g., application layer, data layer, consensus layer, or network layer) and causes (e.g., Ethereum design, smart contract programming, Solidity language, and human factors). While detailed, this classification primarily focuses on specific vulnerability locations, possibly overlooking others such as those in smart contract source code or dependencies.

1.3.2 Smart Contract Code Vulnerability Datasets

Researchers have published a range of unique datasets focused on smart contract vulnerabilities. In this discussion, we examine both the strengths and limitations of these datasets. For instance, Durieux et al. [16] presented two datasets designed to assess the accuracy of smart contract analysis tools. The first dataset includes 69 vulnerable smart contracts annotated with the location and type of vulnerabilities, while the second dataset contains 47,518 raw smart contract samples extracted from Etherscan [12]. Ren et al. [24] compiled a dataset of 46,186 contracts sourced from multiple repositories, including Etherscan, CVE [13], and the Smart Contract Weakness Classification (SWC) [14]. This dataset includes both labeled and unlabeled contracts, with 350 artificially constructed contracts and 214 confirmed vulnerable contracts. Additionally, Zhang et al. [54] developed the Jiuzhou dataset, featuring 176 smart contracts annotated with various vulnerabilities. Each vulnerability type is represented by two contracts: one with the vulnerability and one without. Finally, Furthermore, Gigahorse benchmarks[15] offer a collection of Ethereum smart contracts in both source and binary formats, labeled with corresponding vulnerabilities, some of which originate from Durieux et al. [16].

[12] https://etherscan.io/
[13] https://cve.mitre.org/
[14] https://swcregistry.io/
[15] `https://github.com/nevillegrech/gigahorse-benchmarks`

1.3.3 Prioritizing Vulnerabilities During Code Review

Automatically prioritizing vulnerabilities during reviews is a critical triage and auditing process in software engineering. This approach is increasingly important due to the time-consuming and challenging nature of manually prioritizing code reviews and determining which vulnerabilities to address first. Consequently, researchers have turned to data science techniques to automate this process. For example, Fan et al. proposed binary classifications to decide whether a code review should be addressed or abandoned, while Zhao et al.[65] and Tian et al.[55] utilized multi-class classification to assign priority levels or ranks to code reviews. Various studies have employed different techniques such as Random Forest-based approaches (e.g., Fan et al.[21]), Light Gradient Boosting Machines (e.g., Islam et al.[33]), graph convolutional networks (e.g., Fang et al.[22]), deep multitask learning (e.g., Li et al.[36]), and classical machine learning classifiers (e.g., Valvida-Garcia et al. [57]). These approaches consider multiple features including the type of code vulnerability, its impact, severity, authorship, and more to predict priority levels automatically during software engineering reviews. Moreover, there has been limited research into the automatic prediction of code review and vulnerability priorities within smart contracts.

1.3.4 Detecting Smart Contract Logic Vulnerabilities

Early detection tools for smart contract vulnerabilities have primarily relied on rules and static analysis (such as Slither [19], Smartcheck [48], and Solhint [34]). Another category of tools utilizes symbolic execution, including Mythril [29] and Osiris [49]. Moreover, dynamic analysis tools such as Maian [31] analyze Solidity contracts on a private blockchain during execution. Despite the effectiveness of these tools, an empirical evaluation by Chaliasos et al. [7] reveals that they do not adequately detect logic-related vulnerabilities, which are often the root cause of high-impact attacks. Moreover, Zhang et al. [55] noted that over 80% of exploitable vulnerabilities remain undetectable by automated methods. A study by Zhang et al. [54] defines logic-related vulnerabilities as flaws in decision logic, branching, sequencing, or computational algorithms. These vulnerabilities, found in natural language specifications or implementation languages, are divided into four categories: assembly vulnerabilities, DoS, fairness vulnerabilities, and storage vulnerabilities. Many studies have employed data science techniques such as machine learning and large language models to detect smart contract code vulnerabilities. Sun et al. [45] introduced GPTScan, which combines GPT with static analysis. GPTScan prompts GPT to automatically

recognize scenarios related to logic vulnerabilities and define key variables and statements, which are then verified through static analysis. Jeon [25] utilized a pre-trained BERT model to extract code fragments from smart contracts and identify vulnerable code patterns. Hu et al. [23] utilized GPT-4 to serve as both an auditor for analyzing smart contracts and a critic for reviewing the audits. David et al. [12] investigated the potential of large language models (LLMs) for smart contract security audits by testing them on 52 compromised Decentralized Finance (DeFi) smart contracts. Their findings reveal that while GPT-4 and Claude models correctly identify vulnerabilities in 40% of the cases, they also have a significant false positive rate, necessitating the involvement of manual auditors. Finally, Sun et al. [44] introduced the ASSBert framework based on active and semi-supervised bidirectional encoder representations from transformers (BERT) networks to detect smart contract code vulnerabilities. Current methodologies primarily focus on identifying vulnerabilities in Solidity code, with limited emphasis on detecting vulnerabilities inherent to the contract's logic. Logic-related vulnerabilities, such as sanity checks and inline assembly vulnerabilities, present additional challenges for detection [7, 15].

1.4 Research Method

This section outlines the methodology employed in this PhD dissertation. To comprehensively study smart contract vulnerabilities, it is crucial to incorporate various data sources into our method.

Table 1.1 summarizes each paper along with its respective strategy and collected data.

Table 1.1: Research strategy and data types across this PhD dissertation's papers. Note: SR refers to software repositories

Paper	**Strategy**	**Data**
Paper A	Qualitative Analysis	Unstructured Textual Data: SR, Forums.
Paper B	Repository Mining	Structured Source Code from SR.
Paper C	Data Science	Structured Textual Code Reviews from SR.
Paper D	Data Science	Structured Source Code from SR.

As discussed in Section 1.3, focusing solely on one data source can limit the understanding of vulnerabilities present in real-world smart contracts, po-

tentially leading to significant consequences. Furthermore, analyzing heterogeneous data sources with varying formats requires manual labeling by experts in our methodology, especially in the qualitative analysis conducted in Paper A. Automated analysis alone is inadequate for effectively processing such diverse datasets. Moreover, due to the current limitations of automated analysis tools, human expertise is indispensable to ensure the integrity of our method. Additionally, human expertise remains essential in this research to validate the data before utilizing it in the automation implemented in Papers C and D. Qualitative analysis provides valuable insights for laying the foundation of existing smart contract vulnerabilities. However, as the number of smart contracts continues to grow, relying solely on this qualitative analysis becomes impractical due to its time-consuming and resource-intensive nature. Therefore, various research strategies are employed across the four research papers described in this PhD dissertation. In Paper A, we manually collected data from various software repositories and forums such as Stack Overflow, GitHub, CWE and CVE. Software repositories serve as primary platforms where developers discuss and share their smart contract codes, offering a rich source of real-world smart contracts in their natural form, complete with vulnerabilities, fixes, comments, and textual descriptions. Therefore, they provide a comprehensive and representative overview of smart contract vulnerabilities in practical settings. This aligns with the scope of Paper A, the qualitative analysis paper, as it allows us to gain a holistic understanding of existing vulnerabilities encountered by developers during contract implementation. Consequently, this enables us to identify multiple characteristics and dimensions of these vulnerabilities, contributing to a fundamental and unified taxonomy that integrates our findings with existing taxonomies.

As manually collecting vulnerabilities is labour-intensive, Paper B automates the collection of vulnerabilities and their fixes from software repositories such as GitHub. This automation significantly improves the efficiency of vulnerability tracking for both researchers and developers. Repository mining involves quantitatively analyzing a dataset extracted from a platform hosting structured or semi-structured text, such as a source code repository. Therefore, we use repository mining to collect code fixes implemented by developers, identify the vulnerabilities based on the taxonomy categories proposed in Paper A, and classify them accordingly.

Papers C and D utilize data science research strategies. In data science research, studies analyze software engineering phenomena or artifacts using data-centric analysis methods such as machine learning, computational intelligence approaches, and search-based techniques [51]. In Paper C, we collect data from

code reviews on GitHub and CVE, utilizing the dataset derived from Paper B. We employ machine learning to predict early critical vulnerabilities and prioritize vulnerabilities in these code reviews. In Paper D, we use the vulnerabilities and fixes from Paper B, along with their associated contracts, to automatically detect logic-related vulnerabilities in smart contracts. We then compare the results with the detection of other types of vulnerabilities using machine learning. In addition to achieving the specific aims of each study and the overall objectives of this PhD dissertation, a fundamental goal is to demonstrate a thorough understanding of essential research methods for a PhD. Therefore, across the different papers, we employ a variety of research strategies: qualitative analysis in Paper A, repository mining in Paper B, and data science techniques in Papers C and D. This dissertation employed the mentioned research strategies across four independent papers to achieve the unified goal of improving smart contract security. It also demonstrates proficiency in utilizing diverse data sources and strategies to investigate significant research questions. We detail the research strategies employed in each paper in Chapters 2 to 5.

1.5 Contributions

This section summarizes the four papers included in this PhD dissertation. The full papers can be found in Chapters 2 to 5.

1.5.1 Paper A: Characterizing Smart Contract Code Vulnerabilities.

Code vulnerabilities in smart contracts have resulted in severe financial and reputational damage. Therefore, it is important to understand the nature of code vulnerabilities in Ethereum smart contracts to detect and prevent them in the future. Various vulnerability classification schemes have been proposed in the literature to define types of code vulnerabilities in smart contracts. However, these classifications have several limitations, which can be summarized as follows:

- **L1:** Existing classifications mix different code vulnerability dimensions.
- **L2:** Significant variation in data sources compromises the completeness and representativeness of classifications.

- **L3:** Existing classifications overlook important data sources such as SWC and CVE databases, which are essential for comprehensive sampling and may introduce biases into existing classifications.

These shortcomings highlight the need for deeper analysis within the problem domain to achieve precise classification of smart contract code vulnerabilities. Such classification could enhance the consistency of audits conducted by software engineers and experts, particularly in assessing potential security incidents.

To address these limitations, the goal of Paper A in this PhD dissertation is therefore is two-fold: (i) to characterize vulnerabilities in Ethereum smart contracts written in Solidity, and (ii) to provide an overview of existing classification schemes in relation to this characterization.

To achieve this goal, Paper A answers the following sub-research questions:

- ***RQ1.1*** *What categories of code vulnerabilities appear in smart contracts?*
- ***RQ1.2*** *How do frequency distributions of smart contract code vulnerability categories compare across data sources?*
- ***RQ1.3****What impact do the different categories of smart contract code vulnerabilities have?*

Paper A establishes the groundwork for this PhD dissertation by quantitatively and qualitatively analyzing and categorizing code vulnerabilities in Ethereum smart contracts, reviewing existing classification schemes in light of the proposed characterization, and developing a unified, widely applicable classification scheme. Papers B, C, and D utilize these established categories and definitions of code vulnerabilities as foundational elements for further research and development.

To address the research questions, Paper A utilizes diverse data sources, including two public coding platforms (GitHub and Stack Overflow) and two vulnerability databases (SWC and CVE).

In the paper, we employ keyword-based searches to collect smart contract code vulnerabilities from these selected sources, focusing on terms such as "smart contract," "Solidity," and "Ethereum." It then applies an open card sorting approach to develop a classification scheme for smart contract code vulnerabilities based on their error source and impact. Finally, the paper compares and aligns existing classification schemes with the proposed classification scheme. Figure 1.4 provides an overview of the scope covered in Paper A. It includes details on the various data sources utilized for the analysis, with a breakdown of the data collected from each source.

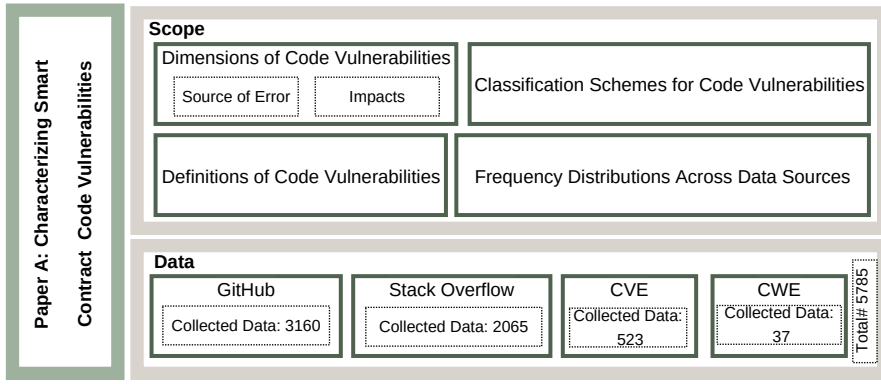


Figure 1.4: Overview of the scope of Paper A and data sources, with data breakdown per source

The results of Paper A include a classification taxonomy comprising 11 categories that describe the sources of code vulnerabilities and 13 categories that outline their potential impacts. The quantitative analysis in Paper A reveals that categories such as language-specific coding and structural data flow are predominant, though their frequency varies significantly across different data sources. These findings provide researchers with a clearer understanding of smart contract code vulnerabilities by defining key dimensions within the problem domain. The taxonomy is further supported by mappings to existing literature-based classifications and frequency distributions across the defined categories.

1.5.2 Paper B: Automatic Identification and Categorization of Smart Contract Code Vulnerabilities and Their Fixes

Studying the code vulnerabilities in paper A revealed several significant gaps in the field, which we detail below:

- **GP1 - Scarcity of Datasets:** There is a notable lack of open, well-structured datasets on smart contract vulnerabilities and their corresponding fixes, which support qualitative and quantitative research.
- **GP2 - Lack of Granularity:** There is no variation in the level of gran-

ularity in labeled datasets.

- **GP3 - Vyper Language Research:** There is a shortage of datasets that support research on the Vyper Ethereum smart contract language.
- **GP4 - Automated Mining and Classification:** There is a lack of approaches that automatically mine, classify, and label smart contract vulnerabilities and their corresponding fixes.

To address these research gaps, Paper B aims to achieve a two-fold goal: (i) develop a framework capable of automatically mining and classifying smart contract vulnerabilities and their corresponding fixes from GitHub and CVE records, leveraging these sources to systematically identify and categorize vulnerabilities within Ethereum smart contracts written in Solidity, thereby enhancing the efficiency and scalability of vulnerability research in blockchain technology; and (ii) create openly accessible datasets containing examples of vulnerable smart contracts and fixes. This dataset serves as a valuable resource for evaluating and benchmarking analysis tools designed to detect and mitigate vulnerabilities in smart contracts, thereby advancing the state-of-the-art in smart contract security analysis.

The key contributions of our research in Paper B can be summarized as follows:

- Developed AutoMESC, an Automated Framework for mining and classifying smart contract vulnerabilities and their corresponding fixes from GitHub and CVE records. It supports a broad range of smart contract vulnerability types, specifically 36 different types.
- Created and Evaluated a Dataset: Using AutoMESC, we devised a dataset that includes a large number of instances of vulnerable smart contract code and their corresponding fixes at different levels of granularity. We conducted a rigorous evaluation of its quality in terms of accuracy, relevance, and provenance with detailed statistics. This dataset enhances the resources available for data-driven research in smart contract vulnerabilities.

In addition to the above contributions, Paper B reviewed available datasets by conducting a comprehensive assessment of existing datasets for smart contract vulnerabilities. This review focused on the two most popular Ethereum languages, Solidity and Vyper, and identified critical gaps and needs within the current datasets.

The AutoMESC framework is structured into three primary phases: Data Collection, Data Preprocessing, and Data Labeling. These phases are fully automated and designed for regular updates to maintain current and precise data. During the Data Collection phase, existing repositories are systematically mined to gather data, which is subsequently stored in a relational database. In the Data Preprocessing phase, the collected data undergoes cleaning and structuring to ensure readiness for analysis. Lastly, the Data Labeling phase employs seven vulnerability detection tools for automated labeling of vulnerabilities within the data, after which the labeled data is stored for future reference.

The Data Collection in AutoMESC is a systematic process designed to automatically mine GitHub projects for vulnerability-fix pairs in Solidity or Vyper using the GitHub API, ensuring regular updates to capture new vulnerabilities and fixes. At the same time, it collects and processes CVE records from the NVD, organizing them chronologically and removing duplicates. The methodology includes detailed filtering of commits related to vulnerabilities based on specific keywords, mapping vulnerable code to fixes at multiple levels of granularity. This automated process operates continuously and is designed to update regularly, providing a reliable foundation for ongoing research in smart contract security.

AutoMESC's Data Preprocessing includes filtering out non-code commits and focusing on those impacting Solidity (.sol) and Vyper (.vy) files. AutoMESC labels vulnerabilities by first collecting output from selected tools for each dataset, identifying detected vulnerability names and their locations (line numbers), and standardizing names based on predefined mappings. Using a majority rule approach, AutoMESC labels a vulnerability if at least 50% of detecting tools agree on its presence at the same position. Additionally, to mitigate noise generated by keyword matching for commits fixing vulnerabilities, AutoMESC verifies fixed versions using selected tools.

The devised dataset in paper B includes essential attributes extracted from CVE such as CVE ID, published date, description, and severity. Each vulnerability is linked via a unique hash to related commits in GitHub repositories, facilitating direct access to source code changes and fixes. The categorization in paper B of vulnerabilities is done based on the taxonomy proposed in paper A and the Common Weakness Enumeration (CWE) types extracted from CVE records, providing detailed metadata including names, descriptions, and URLs for each CWE type associated with a vulnerability.

AutoMESC further enhances granularity by extracting vulnerability-fix pairs at both file and line levels. The dataset contains the code before and after modifications, presented in Git-compatible formats, enabling precise analysis and

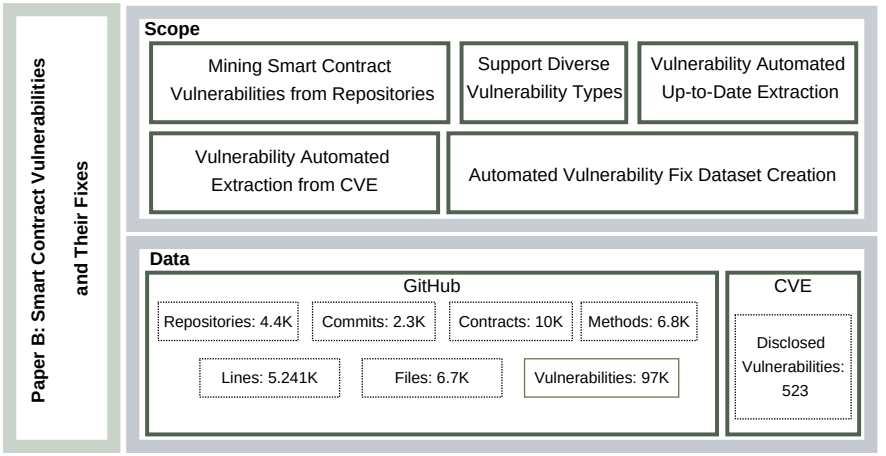


Figure 1.5: Overview of the scope of Paper B and resulting dataset from two main data sources, with data breakdown per source

tracking of vulnerability fixes across different levels of code granularity. These phases collectively advance AutoMESC to systematically mine, categorize, and analyze data critical to advancing research in smart contract security. Figure 1.5 summarizes the scope of Paper B and the resulting dataset with data breakdown per source.

1.5.3 Paper C: Automated Prioritization of Smart Contract Code Vulnerabilities

Due to late-disclosed vulnerabilities, smart contracts are susceptible to numerous security threats. Consequently, it is crucial to automate the prioritization and early prediction of these vulnerabilities to address them in a timely manner. To tackle this challenge, Paper C introduces an automated approach, PrAIoritize, designed to prioritize and predict critical code vulnerabilities in Ethereum smart contracts early during the code review process.

Hence, Paper C answers the following sub-research question:

- ***RQ3.1*** *To what extent can smart contract code vulnerabilities be successfully prioritized during code review processes?*

To address this question, paper C builds on the foundations of paper A and paper B by developing an automated approach called PrAIoritize.

First, paper C quantifies the occurrence of zero-day vulnerabilities—those vulnerabilities that are unknown prior to their disclosure in Ethereum smart contract code— by examining the timing of exploit disclosures in the CVE database. This analysis underscores the need to timely prioritize and discover code vulnerabilities in smart contracts, thus motivating future work on automated triage in this domain.

Next, based on the findings from paper A, a domain-specific lexicon is created, incorporating defined vulnerabilities and their impacts as outlined in paper A's taxonomy. Paper C then employs natural language processing (NLP) techniques and this lexicon to automatically label unlabeled code vulnerabilities in code reviews collected from paper B. Finally, feature engineering is conducted for code reviews and large language models (LLMs) are used to prioritize these vulnerabilities early and predict critical code issues from the code reviews.

Selecting NLP and LLMs to address this problem domain holds significant potential because the structure of smart contracts, which includes sequential code with dependencies among statements, shares similarities with natural language text. Hence, one can view smart contract code as analogous to the natural language used in legal contracts.

Furthermore, another rationale for selecting LLMs is that smart contracts frequently adhere to specific templates and standards, resulting in identifiable patterns and repetitive code structures [10]. This characteristic highlights the utility of LLMs in predicting and prioritizing code vulnerabilities during the code review process.

Our evaluation shows significant improvements over state-of-the-art baselines and well-known LLMs (e.g., T5) in similar classification tasks, showing an increase in F-measure, precision, and recall ranging from 4.82% to 27.94%.

Our findings from Paper C highlight the importance of integrating quantitative and data science methods such as PrAIoritize into software development processes, particularly in smart contract auditing. Secondly, our research emphasizes the ongoing need to explore the dynamic evolution of code vulnerabilities and their implications for security practices. This involves studying how automated prioritization influences vulnerability management strategies and code review efficacy. Thirdly, the findings suggest avenues for future research to develop multi-objective approaches that balance early prediction and comprehensive prioritization in code review processes. Ultimately, these implications emphasize the potential of quantitative and data science methods to advance security practices during software development and engineering. Figure 1.6 pro-

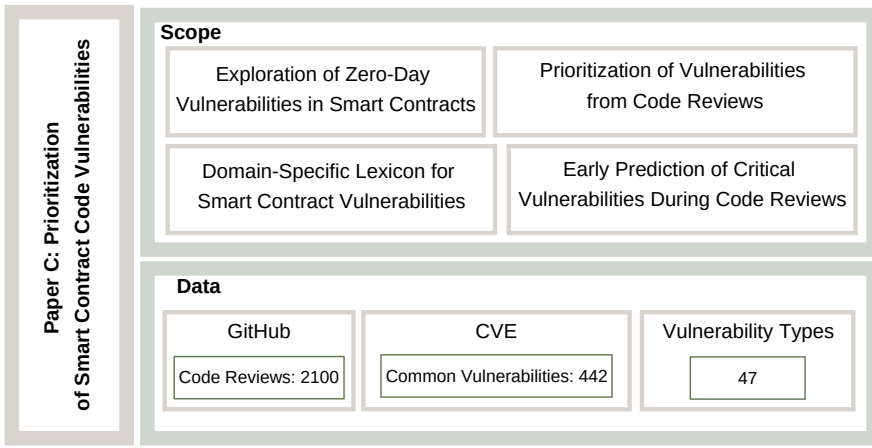


Figure 1.6: Overview of the scope of Paper C and data sources, with data breakdown per source

vides an overview of the scope of Paper C, including a data breakdown per data source used in the paper.

1.5.4 Paper D: Automated Detection of Logic Code Vulnerabilities

Among the categories of code vulnerabilities defined in Paper A, logic code vulnerabilities in smart contracts often are the root cause of high-impact cyberattacks on smart contracts. Furthermore, existing tools frequently exhibit inefficiencies in detecting these types of code vulnerabilities, along with logic errors and sanity checks. Motivated by the findings in Paper C, Paper D aims to enhance the identification and automated detection of logic code vulnerabilities in smart contracts by introducing Sóley, an automated detection approach leveraging Large Language Models (LLMs). Paper D addresses this objective through three sub-research questions:

- ***RQ4.1 (Identification)****: To what extent do historical code changes reveal logic vulnerabilities in smart contracts?*

- ***RQ4.2 (Detection):*** *How can we automatically detect logic vulnerabilities in smart contracts via LLMs?*

- ***RQ4.3 (Mitigation):*** *What specific strategies do developers employ in their code changes to mitigate potential logic vulnerabilities in smart contracts?*

Paper D uses the dataset from Paper B and related smart contracts, clustering code changes into eight groups based on frequent keywords. From each cluster, we sample 10% of code changes and conduct qualitative analysis to identify logic vulnerabilities using open coding methods. The rest of the data is labeled using state-of-the-art tools and regular expressions. Each JSON file details detected vulnerabilities, including their type, contract filename, erroneous lines, and metadata.

Given that Solidity contracts with identical tokens can vary in cleanliness or vulnerability, we select individual lines granularity of code rather than whole contracts. Our study, similar to Wartschinski et al. [51], examines code tokens—keywords, identifiers, operators, and literals—responsible for logic vulnerabilities and their contextual usage. This method helps pinpoint vulnerabilities within sequential code, considering the interdependence of statements. After that, we slice the code into negative slices (lines demonstrating vulnerabilities) and positive slices (lines without vulnerabilities) using a Byte-level BPE tokenizer trained on our corpus with Hugging Face tokenizers [16].

Machine learning-based models, especially transformers, are highly effective in capturing the sequential nature and semantics of code. Given their success in similar tasks, we apply large language models (LLMs) to detect logic vulnerabilities in smart contracts. We randomly selected 1000 samples of five logic vulnerabilities, each containing inline assembly fragments classified as logic vulnerabilities in the literature, to use as training data. Each model was trained on this dataset with consistent settings and hyperparameters, followed by an evaluation benchmark using established metrics.

Paper D's findings show that code changes offer valuable insights into logic vulnerabilities, detecting numerous issues. Analyzing changes alongside contract source code and related contracts is crucial for a comprehensive understanding. Manual analysis of code changes, especially in business logic, is challenging yet uncovers novel vulnerabilities overlooked by automated methods. Logic vulnerabilities in smart contracts vary from syntactic issues detectable by tools or

[16] https://huggingface.co/huggingface/CodeBERTa-small-v1

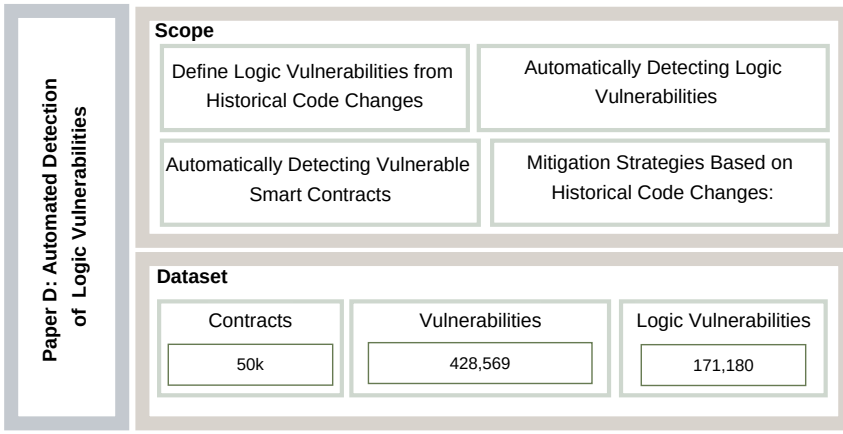


Figure 1.7: Overview of the scope of Paper D and resulting dataset

regular expressions to semantic-specific vulnerabilities requiring a deep understanding of contract semantics and business logic context.

Additionally, Paper D introduces Sóley, an automated detection system for logic-related vulnerabilities, which demonstrates superior accuracy compared to both baseline and state-of-the-art LLMs, achieving a 5%-9% improvement. Our experiments underscore LLMs' effectiveness in identifying various vulnerabilities without extensive feature engineering. However, all models encounter difficulties in detecting Re-entrancy (RE) vulnerabilities, likely due to limited context or examples in the training set. Re-entrancy vulnerabilities occur when a smart contract repeatedly calls an external contract before the initial execution is completed. This can lead to a situation where an attacker can withdraw funds from the contract by exploiting this recursive loop. A notable example is the 2016 DAO attack [31], where a re-entrancy flaw was used to withdraw millions of dollars in Ether. Figure 1.7 overviews Paper D's scope and the resulting dataset.

1.6 Summary of Results

The contributions of this dissertation can be summarized as follows. The first goal, G1, focuses on the identification, characterization, and categorization of

code vulnerabilities in Ethereum smart contracts to understand their types, sources of errors, and impacts. RQ1 and RQ2 represent G1, guiding our exploration and automated mining of code vulnerabilities from diverse sources. The findings related to G1 are summarized in Figure 1.8. RQ1 reveals a significant number of smart contract code vulnerabilities discussed across multiple data sources, highlighting their critical analysis. The resulting frequency distribution in RQ1 indicates that the Language-specific coding category and the Structural data flow category are the most prevalent code vulnerability categories in Ethereum smart contracts.

Regarding code vulnerabilities, RQ1 defined three dimensions of the problem: the source of error, the location at the network level, and the behavior and consequences resulting from an exploit of the code vulnerability. Moreover, concerning impacts, RQ1 defines two dimensions: impact on the software product itself and impact on business factors. We refer to these impact dimensions as *I-D*. I-D1 assesses the impact of code vulnerabilities on the software product itself, such as performance issues in smart contracts from a software engineering perspective. I-D1 assesses the impact of code vulnerabilities on the software product itself, such as performance issues in smart contracts from a software engineering perspective. I-D2 describes the impact on business factors and contract owners, such as financial losses or compromised business information. Given the unique characteristics of smart contracts, such as immutability and gas consumption, addressing severe code vulnerabilities before deployment is crucial. Our research in RQ1 led to the classification of 2143 extracted vulnerabilities into 47 unique smart contract vulnerabilities, grouped into 11 categories. Within our classification scheme, we aligned with existing literature and identified 21 novel code vulnerabilities. Most literature-based classifications for smart contract vulnerabilities typically include broad categories, such as availability.

RQ2 complements RQ1 in achieving G1 by implementing the AutoMESC automated framework for mining and classifying Ethereum smart contract vulnerabilities and their corresponding fixes from GitHub and CVE records in the National Vulnerability Database. AutoMESC was successfully implemented as a fully automated system that mines, classifies, and labels collected code vulnerabilities based on various types of vulnerabilities. Additionally, it gathers metadata crucial for data-intensive research in smart contract security, including vulnerability detection, classification, severity prediction, and automated repair. The resulting dataset initially includes around 97,000 vulnerabilities. Among the most frequently occurring vulnerabilities in AutoMESC data are hard-coded addresses and implicit visibility levels, each with over 10,000 occur-

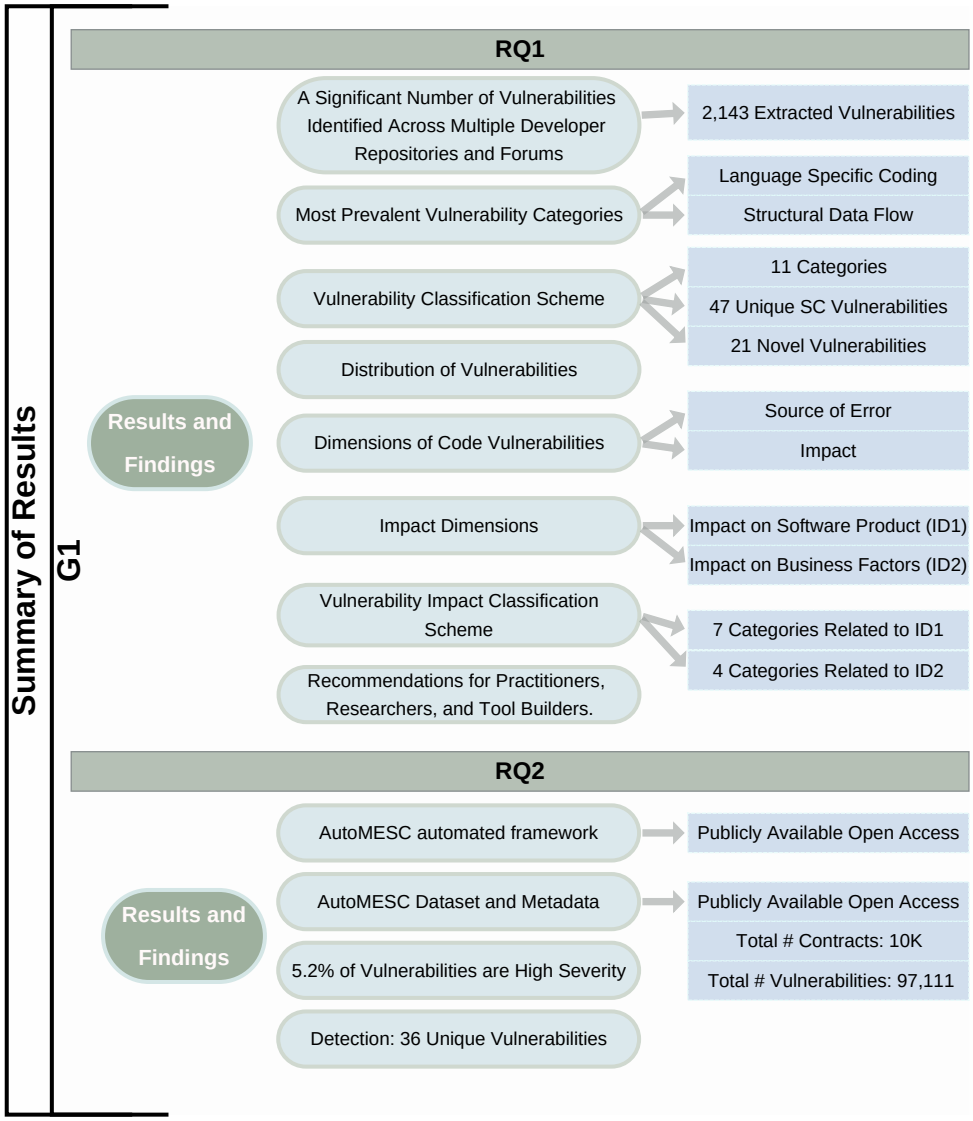


Figure 1.8: Summary of findings and results in Goal 1 (RQ1 and RQ2)

Table 1.2: Top frequently labeled smart contract vulnerability types using AutoMESC

Vulnerability Type	**Total**
Implicit visibility level	12536
Hardcoded address	11633
Upgrade code to Solidity	8431
Compiler version not fixed	7409
Comparison with block.timestamp	5116

rences.

Table 1.2 displays the most frequently labeled smart contract vulnerabilities in AutoMESC data. Furthermore, RQ2 findings show that the majority of vulnerabilities in AutoMESC have low severity levels, with only 5.2% classified as high severity. The evaluation in RQ2 reveals that the AutoMESC dataset offers a variety of attributes suitable for diverse research projects on smart contract code vulnerabilities. Significantly, it addresses the challenge of dataset timeliness by updating regularly to incorporate newly disclosed vulnerabilities. This enhancement boosts the dataset's quality by keeping it up-to-date with the rapid evolution and ongoing developments in smart contract vulnerabilities, effectively adapting to the dynamic nature of these digital assets. The second goal, G2, focused on enhancing automated detection and classification techniques to improve the security of Ethereum smart contracts. This objective was addressed through two research questions, RQ3 and RQ4, the outcomes of which are summarized in Figure 1.9. RQ3 revealed that 77.22% of the analyzed smart contract vulnerabilities were identified as zero-day vulnerabilities, highlighting a significant prevalence of previously unknown vulnerabilities exploited by threat actors. Additionally, smart contract vulnerabilities currently lack dedicated tracking systems, relying instead on community-driven platforms and associated code reviews for disclosure, prioritization, and patching. Furthermore, it also demonstrates that our approach, leveraging data science techniques and large language models, effectively prioritizes code vulnerabilities in smart contract reviews, outperforming both baseline models and state-of-the-art pretrained models.

Finally, the results from RQ4 provide substantial insights into logic vul-

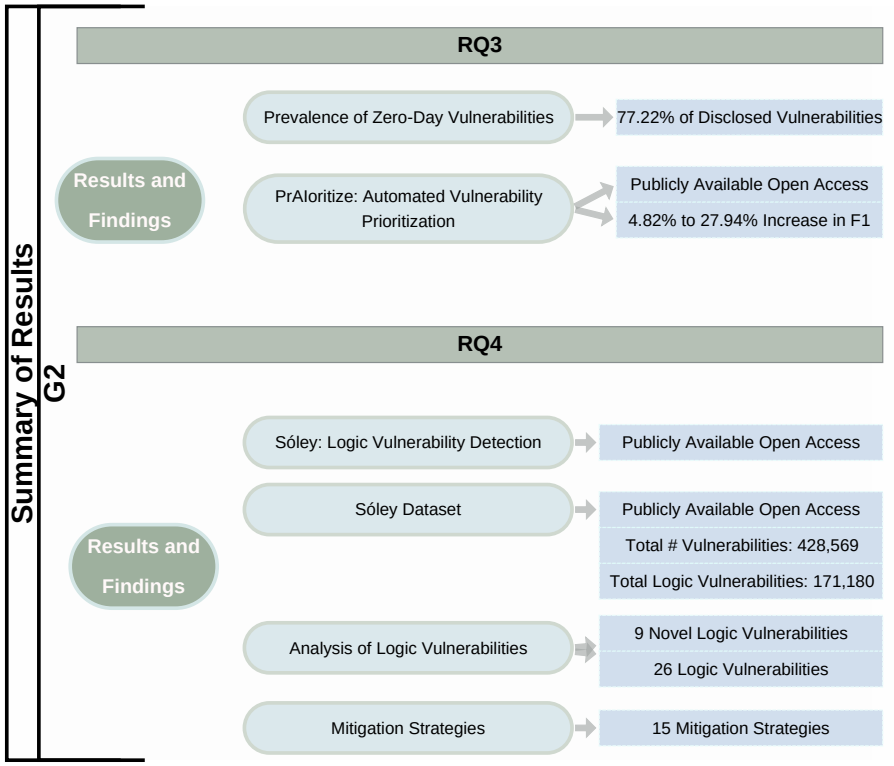


Figure 1.9: Summary of findings and results in Goal 2 (RQ3 and RQ4)

nerabilities within smart contracts, contributing to G2. The analysis of code changes highlighted a range of logic-related issues, emphasizing the necessity of examining both the contract's source code and its interconnected contracts for a comprehensive understanding. Manual analysis of business logic revealed previously unidentified vulnerabilities, underscoring their challenging nature, which automated methods often overlook. These vulnerabilities span from syntactically identifiable flaws, easily pinpointed by tools or regular expressions, to semantic-specific vulnerabilities requiring profound contextual understanding. Our automated detection system, Sóley, demonstrated superior accuracy compared to both baseline models and state-of-the-art large language models, showing an improvement ranging from 5% to 9%. However, challenges persist in effectively detecting vulnerabilities related to regular expressions, likely due to insufficient contextual information or examples in the training set. Additionally, the study outlines 15 mitigation strategies used by developers to address logic vulnerabilities in smart contracts.

1.7 Discussion and Implications

The joint outcomes of the four papers included in this dissertation raise several important discussion points regarding smart contract security and code vulnerabilities. First, there is a need for continuous study of these code vulnerabilities and the regular updating of the unified classification scheme. Secondly, given that software repositories reveal numerous code vulnerabilities in smart contracts, automated approaches to extract and classify these vulnerabilities are necessary. The third discussion point explores the importance of prioritizing code reviews in smart contracts and the role of automated detection methods in enhancing smart contract security. Finally, the outcome of this dissertation suggests significant implications for practitioners, stakeholders, researchers, and tool builders in software engineering to improve the security of smart contracts. We discuss these four points in the following sections.

1.7.1 Analyzing Smart Contract Code Vulnerabilities with Orthogonal Data Sources and Diverse Classification Schemes

Our research reveals a significant number of smart contract code vulnerabilities discussed across social coding platforms and vulnerability repositories, meaning there is a critical need for continuous and up-to-date analysis of code vulnera-

bilities in this area. The distinctive characteristics of smart contracts emphasize the necessity of addressing potentially severe code vulnerabilities prior to deployment, such as logic code vulnerabilities that are mostly the reason for high-impact attacks.

Existing classification schemes often have limitations, focusing narrowly on single dimensions or combining multiple dimensions without clear integration. Our work unifies these dimensions through mapping to provide a clearer understanding of how different classification schemes relate and interact.

Moreover, relying on broad categories such as security and availability may lack the granularity needed for precise reasoning and fail to ensure category consistency. For instance, in the event of an incident, security engineers should be able to precisely identify and reason about the specific source of error within these categories in order to detect it and fix it. Our analysis of frequency distributions across diverse data sources reveals the biases inherent in studies confined to single sources and the disparities between established databases (e.g., CVE, SWC) and community-driven platforms (e.g., Stack Overflow, GitHub). Additionally, our examination of smart contract code vulnerabilities assesses their impacts across dimensions such as unexpected stoppage, memory disclosure, and data corruption. These findings are crucial for devising effective strategies to prioritize, prevent, detect, and mitigate vulnerabilities in smart contracts and blockchain applications.

Implications from our findings suggest pathways to improve smart contract development and security practices. Firstly, smart contract detection tools should prioritize categories of code vulnerabilities identified in our research, especially those underrepresented in established databases such as SWC and CVE, which can enhance detection capabilities and mitigate overlooked vulnerabilities. Secondly, collaboration between software engineers and Solidity language designers is crucial to define and implement patch templates for known categories of code vulnerabilities. These templates can facilitate the process of addressing vulnerabilities based on their specific root causes, thereby improving code quality and security. Furthermore, clear and accessible coding best practices for smart contracts should be disseminated to developers. Aligning these guidelines with recommendations from Solidity language designers can mitigate language-specific coding vulnerabilities and optimize contract performance, particularly in areas such as gas consumption and function usage. Automated solutions are essential for prioritizing the resolution of smart contract code vulnerabilities based on their potential impacts. By focusing on vulnerabilities with the highest risk, developers can effectively prevent financial losses and enhance overall contract security. Maintaining the relevance and accuracy of taxonomies used

for classifying smart contract vulnerabilities requires continuous updates and expert reviews. Engaging developers, auditors, and researchers in this process ensures that taxonomies evolve to address emerging vulnerabilities and reflect real-world scenarios effectively.

To sum up, our research identifies prevalent smart contract code vulnerabilities from orthogonal data sources, revealing significant novel vulnerabilities not extensively discussed in existing literature. We also shed light on the limitations of current classification schemes and underscore the impacts of these vulnerabilities. Addressing isolated aspects or dimensions of code vulnerabilities as orthogonal perspectives hinders comprehensive understanding, complicating the pinpointing and effective resolution of these issues.

1.7.2 The Importance of Timely Classification and Automated Detection of Smart Contract Code Vulnerabilities

Effective code review processes are pivotal in smart contract development to ensure their security. GitHub, a well-known platform for collaborative software development, provides valuable insights through code changes (diffs) during reviews. These diffs encapsulate discussions, patches, and comments, offering a dynamic view of how vulnerabilities evolve and are addressed throughout the development lifecycle. Our findings successfully yielded a framework capable of automatically mining, classifying, and labeling smart contract vulnerabilities and corresponding fixes from GitHub[17] and CVE[18] records.

This framework extracts a set of attributes related to code vulnerabilities, significantly enhancing the security of smart contracts. For instance, these attributes can help in developing more accurate detection, improving the effectiveness of vulnerability management strategies, and enabling timely security measures such as Just-in-Time defect detection. Furthermore, the resulting datasets can be utilized in various use cases that rely on large volumes of labeled data, such as machine-learning (ML) applications. Continued exploration of these datasets can lead to advancements in automated vulnerability detection, real-time threat intelligence, and informed decision-making in smart contract development.

Our findings imply that early prioritization and detection significantly benefit practitioners by enhancing smart contract development and auditing pro-

[17] https://github.com/
[18] https://cve.mitre.org/

cesses. By prioritizing code reviews more effectively, critical code vulnerabilities are promptly addressed which may reduce security breaches. Automated triage accelerates the identification of critical vulnerabilities, minimizing the risk of overlooking urgent issues. Early prediction of critical vulnerabilities enables developers to recognize and address potential risks swiftly. Third-party auditors can focus on high-impact areas, providing valuable insights to stakeholders. This approach also improves patch management by categorizing vulnerabilities based on priority, reducing development time and costs.

Early prioritization and detection can also advance the understanding of smart contract code vulnerability evolution and their priorities. It facilitates studies on the correlation between developer attributes, review methodologies, and code vulnerability prioritization. By leveraging early detection, researchers can develop multi-objective approaches to address early code review prediction and prioritization effectively. Additionally, this method allows for a deeper investigation into how socio-technical factors influence smart contract security. Moreover, early prioritization and detection can be integrated into broader security frameworks and tools within smart contract development. These tools can predict and prioritize code review requests and integrate them into existing development systems such as Gerrit to enhance the efficiency of the review process. This integration particularly benefits project maintainers who oversee large codebases with numerous pending reviews. This integration ensures that vulnerabilities are addressed early in the process, minimizing the potential for exploits

1.7.3 Distinctive Features of Logic Vulnerability Detection

Detecting logic vulnerabilities in smart contracts presents unique challenges compared to other types of vulnerabilities. Existing tools often struggle to effectively identify these vulnerabilities due to several reasons. First, logic vulnerabilities involve intricate decision-making processes within the code that cannot be adequately captured by simple regular expressions alone. These vulnerabilities often stem from complex logical errors rather than straightforward patterns, making them harder to pinpoint automatically.

Empirical evaluations of smart contract security tools highlight significant shortcomings in detecting logic-related issues. Many tools tend to produce numerous false positives, flooding users with irrelevant alerts and potentially masking real threats. Researchers, including Chaliasos et al., emphasize the inefficiency of current tools in identifying logic-related vulnerabilities, such as

flaws in sanity checks or nuanced logic errors. These vulnerabilities are critical as they can lead to substantial financial losses or exploitation of unintended behaviors in decentralized applications.

Moreover, while state-of-the-art tools offer robust capabilities, they may not cover the full range of logic vulnerabilities. Complex variants of these vulnerabilities may escape detection, further highlighting the limitations of relying solely on automated tools for comprehensive security assurance. Consequently, a combined approach that integrates automated tools with human oversight, such as manual validation and contextual understanding, becomes essential.

Manual inspection plays a pivotal role in uncovering new or uncommon logic vulnerabilities that automated methods might overlook. However, this manual process is labor-intensive and lacks systematic verification mechanisms, posing challenges in scaling and ensuring consistency across large datasets or complex smart contracts. As a result, while automated tools can efficiently handle well-known vulnerability types such as locked ether or incorrect equality checks, they may struggle with more nuanced logic-related issues that require deeper contextual understanding and domain expertise.

In summary, the detection of logic vulnerabilities differs significantly from other types of vulnerabilities in smart contracts due to their complexity, variability, and the limitations of current automated detection methods.

Therefore, this area still requires further investigation, and a domain-specific dataset for logic vulnerabilities related to business logic is still needed to enhance detection capabilities.

Moreover, it is important to acknowledge the limitations inherent in focusing on Ethereum and Solidity smart contracts in our research. Therefore, our results cannot be generalized to other blockchain platforms, and adapting our results to other programming languages and blockchain platforms requires careful consideration. Different blockchain platforms may have unique architectures, consensus algorithms, and programming paradigms that can influence the nature and severity of smart contract vulnerabilities. As a result, while our findings provide valuable insights within the Ethereum context, further research is needed to explore the applicability and relevance of our results in diverse blockchain platforms. We encourage future studies to build upon our work by examining smart contract vulnerabilities across a broader spectrum of languages and platforms, adapting classification schemes accordingly. Furthermore, our findings are tied to the current state of the Solidity language and its runtime environment on the EVM. As new smart contract languages emerge, they introduce various design and security models that can significantly impact smart contract vulnerabilities. Consequently, the results obtained from Solidity

may not directly translate to other languages that incorporate different features, such as formal verification or alternative memory and data structures. Lastly, our methods can be replicated using data from other blockchain platforms and programming languages, which may yield a different range of vulnerabilities, classification schemes, and various detection results.

1.8 Validity Threats

This section discusses the challenges to the validity as well as the reliability of the results obtained in the PhD dissertation. To identify threats to validity, we used the standard methodology proposed by Feldt et al. [20]. We have mapped these threats to their corresponding research questions (RQs) and the respective chapters in this dissertation, specifically Chapters 2 to 5.

1.8.1 Internal Validity

In several parts of this PhD dissertation, we used manual labeling when automated labeling was not feasible, particularly for qualitative analysis of unstructured text and code data. For instance, in Chapter 2, addressing RQ1, we used open card sorting to label code vulnerabilities from diverse and heterogeneous data sources. In Chapter 5, addressing RQ4, we applied an open coding scheme to identify logic code vulnerabilities in historical code changes. Given the subjective nature of these methods and the potential for bias, two experts independently verified the manual labeling for both RQ1 and RQ4. In cases of disagreement, the experts discussed until they reached a consensus. The labeling process was time-intensive. While we did not account for fatigue factors, we relied on the experts working in shorter iterations to mitigate this issue. However, there remains a potential threat to validity that fatigue might have affected the quality of the labels. To further minimize this threat, we employed and reported the Kappa coefficient for RQ1. For RQ4, we used regular expressions and state-of-the-art tools to reduce the manual sample size and ensure accuracy.

This leads us to another threat to internal validity: the labeling of code vulnerabilities in large quantities, which can be time- and resource-intensive. This challenge was particularly pronounced in Chapter 2 (RQ2) and Chapter 5 (RQ4), where a substantial amount of labeled code vulnerabilities was necessary. To mitigate the risk of inaccurately labeling code vulnerabilities, we employed established tools for both research questions. In RQ2, we used a majority ap-

proach to validate the results, while in RQ4, we verified outputs using regular expressions to align and minimize the threat of false positives or negatives. However, it is important to acknowledge that no single tool for code vulnerability detection is entirely reliable. Different tools may produce varying numbers of false positives and negatives. To mitigate this, we utilized a majority approach across multiple tools sourced from the literature. This method helps balance out individual biases but may be compromised if many tools exhibit inconsistencies. Additionally, it can be overly conservative, potentially resulting in excessively cautious outcomes that may overlook more nuanced or innovative insights.

In addressing these challenges, we suggest integrating additional tools or employing manual verification methods, such as crowdsourcing, in future work. While these strategies require considerable time and resources, they can be implemented gradually.

Another concern regarding internal validity relates to the selection of specific machine learning models, as observed in Chapter 4 (RQ3) and Chapter 5 (RQ4). To address this potential concern, our design choices were guided by the specific problems addressed in each research question, with the thought of maintaining simplicity and effectiveness in our designs. For example, in both RQ3 and RQ4, we opted to use large language models to capture the semantics and context of code reviews and code, respectively. While other data science techniques may excel in similar tasks, our objective was to analyze code within its contextual framework (e.g., surrounding lines), operating under the assumption that a token signifies a code vulnerability based on its specific usage context. Therefore, we incorporated the consideration of surrounding tokens and lines into our approach. Another example includes our use of tokenization in both research questions. While techniques such as control flows (Abstract Syntax Trees - ASTs) or data graph flow analysis could potentially offer advantages in these tasks, we chose tokenization due to its simplicity and minimal need for feature engineering. This approach allows us to effectively address the research questions while maintaining straightforward and efficient methodologies.

Furthermore, another critical concern for internal validity could arise from potential implementation bugs in the codebase due to the complexity of the developed models, tools, and scripts provided by this PhD dissertation. These include AutoMESC (addressed in RQ2, Chapter 3), Sóley (addressed in RQ4, Chapter 5), and PrAIoritize (addressed in RQ3, Chapter 4). To mitigate this concern, we conducted thorough testing of all these models, scripts, and tools.

Lastly, detailed threats to the internal validity of the included publications are discussed in their respective chapters.

1.8.2 Construct Validity

The construct validity in this PhD dissertation is influenced by three primary challenges. The first challenge pertains to the choice of evaluation metrics used in our study. To mitigate this concern, we opted for widely recognized metrics such as precision, recall, and F-measure, which have been extensively utilized in previous research and formed the basis for our selected baselines in RQ3 and RQ4. Moreover, in RQ2, we evaluated the dataset using the data quality taxonomy introduced by Bosu and Macdonell [5]. This taxonomy assesses data quality along three dimensions: accuracy, relevance, and provenance, and is well-established in the literature.

To address concerns about evaluation fairness across our approaches and selected baselines, we maintained consistency by using identical hyperparameters, training datasets, optimizers, and the same machine for all experiments. Another consideration for the construct validity of our study is the imbalanced distribution of code vulnerabilities across the dataset. To ensure fair classification in RQ4, despite varying occurrences of different code vulnerabilities, we standardized the number of instances across all categories. Furthermore, the high performance observed across classification categories in RQ3 indicates that the imbalance in data did not significantly impact the effectiveness of our approach.

Third, the domain-specific lexicon construction in RQ3 considers known types of code vulnerabilities and was developed based on keywords associated with these types and their impacts for assigning classification categories. To ensure accuracy, an expert reviewed the lexicon and keywords. Furthermore, the construction of classification categories in RQ3 and RQ4 does not adhere to standardized classifications due to the lack of standardization in smart contracts. However, these categories were identified based on code vulnerability types extracted from the literature and prioritized similarly to state-of-the-art knowledge. To mitigate this issue, the selected categories were chosen following keyword extraction from various sources such as CVE records, related literature, and Ethereum official documents. In practice, these selected categories can serve as a starting point and may require adjustment based on future developments in the field.

1.8.3 External Validity

External validity in this PhD dissertation is influenced by three primary concerns. First, focusing exclusively on Solidity and the Ethereum blockchain may

limit the generalizability of our findings. Ethereum's unique characteristics, such as its reliance on Proof of Work (PoW) or Proof of Stake (PoS) for consensus, as well as its inherent transparency and immutability, significantly influence the types of vulnerabilities prevalent on the platform. Other blockchains and smart contract languages, such as those used in Hyperledger or different blockchain platforms, may exhibit different or additional code vulnerabilities due to variations in architecture, consensus mechanisms, and execution environment.

Consequently, our results in RQ1, RQ2, RQ3, and RQ4 may not fully capture the breadth of vulnerabilities across all smart contract platforms. However, by replicating our approaches on datasets from different blockchain platforms, researchers can carefully adapt our findings to reflect the unique security challenges present in those platforms. This may allow for a broader understanding of smart contract vulnerabilities, even if the initial findings are derived from Ethereum.

Second, our results may not extend to future versions of Solidity or other smart contract languages. Ethereum undergoes continuous updates, including through hard forks and Ethereum Improvement Proposals (EIPs), which introduce new features and improve language efficiency. These updates can introduce new vulnerabilities or resolve existing ones, impacting the generalizability of our findings over time.

Thirdly, certain identified vulnerabilities may be subject to interpretation. For instance, 'locked Ether contracts' are contracts that accept Ether without providing a means to withdraw or utilize the funds, typically viewed as a vulnerability. However, in cases where the contract is designed to securely hold funds until a specified condition is met, such as in secure savings contracts, this behavior may be intentional and necessary. To mitigate this concern, we provided clear definitions for each vulnerability, specifying the conditions under which they may lead to unexpected outcomes. Another concern arises when vulnerabilities are interconnected, potentially leading to unpredictable behavior. Exploring these interactions presents a future research opportunity to better understand the combined impact of these vulnerabilities.

In summary, while our study provides valuable insights into Solidity and current Ethereum smart contracts, its external validity is contingent on these platforms and their current state.

1.8.4 Reliability

To ensure the reliability of our results, we provided a detailed description of our data collection and analysis processes. We systematically computed inter-

rater reliability coefficients across multiple iterative rounds to establish clear categories and ensure agreement between two expert raters. Additionally, there is a concern about the reliability of the quantitative analysis, considering the potential presence of unidentified vulnerabilities within the contracts. To address this, two experts reviewed the identified vulnerabilities from the manual labeling process. Furthermore, we have made the raw data openly accessible, allowing other researchers and users to validate the results. Moreover, to ensure transparency and reproducibility, we have made all the developed tools, models, scripts, and datasets publicly available. This accessibility allows researchers and developers to verify and validate their functionality independently. These measures collectively strengthen the reliability of our study and facilitate its replication.

1.9 Conclusion and Future Work

With this dissertation, we contribute to the body of knowledge in smart contracts and blockchain security. We do so by following a broad approach that employs repository mining and qualitative methods, extending our understanding of smart contract code vulnerabilities and characterizing them (RQ1), mining and classifying smart contract code vulnerabilities and their fixes (RQ2), prioritizing smart contract code vulnerabilities (RQ3), as well as detecting logic code vulnerabilities in smart contracts (RQ4). Generally speaking, because of the challenging characteristics of smart contracts inherited from blockchain, it is difficult to improve their security post-deployment. Therefore, enhancing their security pre-deployment is imperative. This requires a deep understanding of the available code vulnerabilities in smart contract code across various software repositories and development platforms. The first research question defines 47 critical code vulnerabilities and analyzes them based on two main dimensions: error source and impact. We then categorize the extracted code vulnerabilities by devising a unified classification taxonomy of smart contract code vulnerabilities, involving 11 categories describing the error source and 13 categories describing potential impacts. Moreover, there is a lack of open datasets on smart contract vulnerabilities and their fixes that support research in this area, and the available datasets are limited in various ways, including not being updated regularly. In the labeled datasets, fixes are not addressed, and there is no variation in the level of granularity. Therefore, RQ2 presents AutoMESC, an automated tool for mining and classifying smart contract vulnerabilities and corresponding fixes from GitHub and CVE records. It also presents a dataset with

vulnerability-fix pairs, containing around 97K vulnerabilities and 6.7K fixes. A total of 10K contracts had vulnerability-fix changes. Finally, this dissertation further improves the security of smart contracts by devising Prioritize to automatically prioritize smart contract code vulnerabilities during the code review process and Solely to automatically detect logic code vulnerabilities that could compromise the contract's integrity and functionality. In summary, this PhD dissertation provides a comprehensive framework for understanding, classifying, prioritizing, and detecting smart contract vulnerabilities. The datasets and tools developed here aim to improve the overall security of smart contracts. Given the current state of smart contract security, we see several directions for future research. First, future studies should investigate whether our classification scheme can be generalized by examining other smart contracts in Ethereum languages, such as Vyper. Potentially, our classification might need to be modified to fit other languages. Moreover, more dimensions and characteristics of the studied code vulnerabilities can be explored in the future as more data becomes available. Based on our results, future work should investigate in more depth why frequency distributions differ across data sources. Additionally, a valuable contribution would be to study how the logic of smart contracts performs at runtime by devising sandboxes that allow comprehensive testing. This can help investigate the ways in which defined code vulnerabilities can be exploited. This can be accomplished by developing automated tests or manually testing the contracts inside the sandbox.

Future research should involve collaboration with the broader blockchain and smart contract development communities, including stakeholders and practitioners, to create standardized security practices during development, which could foster a more secure smart contract and blockchain environment overall. Especially considering that some resources, such as the Decentralized Application Security Project (DASP), were last updated five years ago, and the Smart Contract Weaknesses Classification (SWC) is no longer maintained, it is crucial to devise approaches to maintain up-to-date code vulnerabilities and security measures in smart contract resources.

Bibliography

[1] Attacker pockets $10 million from poly network security attack: Beosin. URL https://www.theblock.co/post/237452/attacker-pockets-10-million-from-poly-network-security-attack\beosin 2

[2] Decentralized application security project (dasp). https://dasp.co/. Accessed: July 4, 2024 3

[3] Eip 1202: Token standard. https://eips.ethereum.org/EIPS/eip-20. [Online] Accessed: 01-April-2020 7

[4] National vulnerability database (nvd). https://nvd.nist.gov/vuln. Accessed: July 4, 2024 3, 149

[5] Smart contract weakness classification and test cases. https://smartcontractsecurity.github.io/SWC-registry 3

[6] Governmental contract. https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck (2016) 2

[7] Alharby, M., Van Moorsel, A.: Blockchain-based smart contracts: A systematic mapping study. arXiv preprint arXiv:1710.06372 (2017) 4, 54, 62, 65, 75

[8] Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts. IACR Cryptology ePrint archive **2016**, 1007 (2016) 3, 13, 54, 62, 65, 75, 80, 85

[9] Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). In: International conference on principles of security and trust, pp. 164–186. Springer (2017) 1, 53, 89, 126, 150, 194, 196, 215

[10] Bosu, M.F., MacDonell, S.G.: A taxonomy of data quality challenges in empirical software engineering. In: 2013 22nd Australian Software Engineering Conference, pp. 97–106. IEEE (2013) 39, 132

[11] Brummer, C.: Disclosure, dapps and defi. Stan. J. Blockchain L. & Poly **5**, 137 (2022) 7, 8, 10

[12] Buterin, V., et al.: A next-generation smart contract and decentralized application platform. white paper **3**(37), 2–1 (2014) 1, 119, 191

[13] Canfora, G., Di Sorbo, A., Laudanna, S., Vacca, A., Visaggio, C.A.: Profiling gas leaks in solidity smart contracts. arXiv preprint arXiv:2008.05449 (2020) 7

[14] Chaliasos, S., Charalambous, M.A., Zhou, L., Galanopoulou, R., Gervais, A., Mitropoulos, D., Livshits, B.: Smart contract and defi security tools: Do they meet the needs of practitioners? In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, pp. 1–13 (2024) 15, 16, 145, 191, 195, 203, 205, 213, 224, 225, 226

[15] Chaliasos, S., Gervais, A., Livshits, B.: A study of inline assembly in solidity smart contracts. Proceedings of the ACM on Programming Languages **6**(OOPSLA2), 1123–1149 (2022) 16

[16] Chen, H., Pendleton, M., Njilla, L., Xu, S.: A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. ACM Computing Surveys (CSUR) **53**(3), 1–43 (2020) 3, 14, 62, 63, 65, 71, 89, 193

[17] Chen, J., Xia, X., Lo, D., Grundy, J., Luo, X., Chen, T.: Defining smart contract defects on ethereum. IEEE Transactions on Software Engineering (2020) 3, 13, 53, 54, 63, 66, 75, 86, 90, 92, 178

[18] Clack, C.D., Bakshi, V.A., Braine, L.: Smart contract templates: foundations, design landscape and research directions. arXiv preprint arXiv:1608.00771 (2016) 1, 24, 146, 192

[19] David, I., Zhou, L., Qin, K., Song, D., Cavallaro, L., Gervais, A.: Do you still need a manual smart contract audit? arXiv preprint arXiv:2306.12338 (2023) 16, 225

[20] Di Sorbo, A., Laudanna, S., Vacca, A., Visaggio, C.A., Canfora, G.: Profiling gas consumption in solidity smart contracts. Journal of Systems and Software **186**, 111193 (2022) 2

[21] Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P.: Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 530–541 (2020) 14, 67, 119, 121, 122, 134, 135, 158, 196, 226

[22] Elsden, C., Manohar, A., Briggs, J., Harding, M., Speed, C., Vines, J.: Making sense of blockchain applications: A typology for hci. In: Proceedings of the 2018 chi conference on human factors in computing systems, pp. 1–14 (2018) 1

[23] Ethereum: Solidity by example (2021). URL `https://docs.soliditylang.org/en/latest/solidity-by-example.html` xiii, 8, 60, 105

[24] Ethereum: Introduction to ethereum (Accessed: 2024). URL `https://ethereum.org/en/developers/docs/intro-to-ethereum/` 5, 6

[25] Ethereum Docs: Ethereum accounts (2023). URL `https://ethereum.org/en/developers/docs/accounts/` 5, 193

[26] Fan, Y., Xia, X., Lo, D., Li, S.: Early prediction of merged code changes to prioritize reviewing tasks. Empirical Software Engineering **23**, 3346–3393 (2018) 15, 147, 177

[27] Fang, S., Tan, Y.s., Zhang, T., Xu, Z., Liu, H.: Effective prediction of bug-fixing priority via weighted graph convolutional networks. IEEE Transactions on Reliability **70**(2), 563–574 (2021) 15, 178

[28] Feist, J., Grieco, G., Groce, A.: Slither: a static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), pp. 8–15. IEEE (2019) 15, 128, 191, 195, 199, 224

[29] Feldt, R., Magazinius, A.: Validity threats in empirical software engineering research-an initial survey. In: Seke, pp. 374–379 (2010) 37, 226

[30] Feng, C., Niu, J.: Selfish mining in ethereum. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pp. 1306–1316. IEEE (2019) 6

[31] Hu, S., Huang, T., İlhan, F., Tekin, S.F., Liu, L.: Large language model-powered smart contract vulnerability detection: New perspectives. arXiv preprint arXiv:2310.01152 (2023) 16, 146, 192, 225

[32] Huang, Y., Bian, Y., Li, R., Zhao, J.L., Shi, P.: Smart contract security: A software lifecycle perspective. IEEE Access **7**, 150184–150202 (2019) 13, 61, 62

[33] Institute, N.R.: Survey on blockchain technologies and related services (2016) 1

[34] Islam, K., Ahmed, T., Shahriyar, R., Iqbal, A., Uddin, G.: Early prediction for merged vs abandoned code changes in modern code reviews. Information and Software Technology **142**, 106756 (2022) 15, 177

[35] ISO/IEC: International standard-iso/iec 23643:2020 software and systems engineering — capabilities of software safety and security verification tools (2020) 3

[36] Jeon, S., Lee, G., Kim, H., Woo, S.S.: Smartcondetect: Highly accurate smart contract code vulnerability detection mechanism using bert. In: KDD Workshop on Programming Language Processing (2021) 16, 225

[37] Kalra, S., Goel, S., Dhawan, M., Sharma, S.: Zeus: analyzing safety of smart contracts. In: Ndss, pp. 1–12 (2018)

[38] Kim, S.: Measuring ethereum's peer-to-peer network. Ph.D. thesis, University of Illinois at Urbana-Champaign (2017) 6

[39] Li, Y., Che, X., Huang, Y., Wang, J., Wang, S., Wang, Y., Wang, Q.: A tale of two tasks: Automated issue priority prediction with deep multi-task learning. In: Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 1–11 (2022) 15, 178

[40] Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pp. 254–269. ACM (2016)

[41] Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pp. 254–269 (2016) 7, 145

[42] Marcus, Y., Heilman, E., Goldberg, S.: Low-resource eclipse attacks on ethereum's peer-to-peer network. Cryptology ePrint Archive (2018) 6

[43] Mueller, B.: Smashing ethereum smart contracts for fun and real profit. In: 9th Annual HITB Security Conference (HITBSecConf) (2018) 15, 128, 225

[44] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008) 1, 191

[45] Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th annual computer security applications conference, pp. 653–663 (2018) 15, 128, 225

[46] Praitheeshan, P., Pan, L., Yu, J., Liu, J., Doss, R.: Security analysis methods on ethereum smart contract vulnerabilities: a survey. arXiv preprint arXiv:1908.08605 (2019) 14, 62, 63

[47] Rahimian, R., Clark, J.: Tokenhook: Secure erc-20 smart contract. arXiv preprint arXiv:2107.02997 (2021) xiii, 5

[48] Ren, M., Yin, Z., Ma, F., Xu, Z., Jiang, Y., Sun, C., Li, H., Cai, Y.: Empirical evaluation of smart contract testing: what is the best choice? In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 566–579 (2021) 14, 119, 121, 122, 134, 135

[49] Samreen, N.F., Alalfi, M.H.: A survey of security vulnerabilities in ethereum smart contracts. arXiv preprint arXiv:2105.06974 (2021) 14, 63

[50] Soud, M., Helgason, S., Hjálmtỳsson, G., Hamdaqa, M.: Trustvote: On elections we trust with distributed ledgers and smart contracts. In: 2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS), pp. 176–183. IEEE (2020) 1

[51] Standards, S.E.: Data science empirical standards (2021). URL `https://www2.sigsoft.org/EmpiricalStandards/docs/standards?standard=DataScience`. Accessed: 2024-07-03 17

[52] Sun, X., Tu, L., Zhang, J., Cai, J., Li, B., Wang, Y.: Assbert: Active and semi-supervised bert for smart contract vulnerability detection. Journal of Information Security and Applications **73**, 103423 (2023) 16, 204, 225

[53] Sun, Y., Wu, D., Xue, Y., Liu, H., Wang, H., Xu, Z., Xie, X., Liu, Y.: Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pp. 1–13 (2024) 15, 199, 205, 206, 225

[54] Szabo, N.: Smart contracts (1994). URL `https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html` 7

[55] Tian, Y., Lo, D., Xia, X., Sun, C.: Automated prediction of bug report priority using multi-factor analysis. Empirical Software Engineering **20**, 1354–1383 (2015) 15, 178, 179

[56] Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y.: Smartcheck: Static analysis of ethereum smart contracts. In: 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), pp. 9–16. IEEE (2018) 15, 128, 191, 195, 196, 200, 224

[57] Torres, C.F., Schütte, J., et al.: Osiris: Hunting for integer bugs in ethereum smart contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference, pp. 664–676. ACM (2018) 15, 128, 195, 225

[58] Valdivia Garcia, H., Shihab, E.: Characterizing and predicting blocking bugs in open source projects. In: Proceedings of the 11th working conference on mining software repositories, pp. 72–81 (2014) 15, 178

[59] Wang, W., Hoang, D.T., Hu, P., Xiong, Z., Niyato, D., Wang, P., Wen, Y., Kim, D.I.: A survey on consensus mechanisms and mining strategy management in blockchain networks. Ieee Access **7**, 22328–22370 (2019) 6

[60] Wartschinski, L., Noller, Y., Vogel, T., Kehrer, T., Grunske, L.: Vudenc: vulnerability detection with deep learning on a natural codebase for python. Information and Software Technology **144**, 106809 (2022) 26, 198, 201

[61] Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014) 2, 5, 6, 7, 145, 191, 193, 194

[62] Zhang, P., Xiao, F., Luo, X.: A framework and dataset for bugs in ethereum smart contracts. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 139–150. IEEE (2020) 3, 13, 14, 15, 53, 54, 62, 63, 64, 65, 75, 90, 92, 119, 121, 122, 134, 135, 178, 199, 203, 205, 206, 213, 224

[63] Zhang, Z., Zhang, B., Xu, W., Lin, Z.: Demystifying exploitable bugs in smart contracts. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 615–627. IEEE (2023) 2, 15, 199, 224, 225

[64] Zhao, G., da Costa, D.A., Zou, Y.: Improving the pull requests review process using learning-to-rank algorithms. Empirical Software Engineering **24**, 2140–2170 (2019) 15, 177

[65] Zheng, Z., Xie, S., Dai, H.N., Chen, X., Wang, H.: Blockchain challenges and opportunities: A survey. International journal of web and grid services **14**(4), 352–375 (2018) 5, 193

Chapter 2

Characterizing Smart Contract Vulnerabilities

A Fly in the Ointment: An Empirical Study on the Characteristics of Ethereum Smart Contract Code Weaknesses

Soud, M., Liebel, G., and Hamdaqa, M. 2024.

Empirical Software Engineering, 29(1), p.13.

Abstract

Context: Smart contracts are programs that are automatically executed on the blockchain. Code weaknesses in their implementation have led to severe loss of cryptocurrency. It is essential to understand the nature of code weaknesses in Ethereum smart contracts to prevent them in the future. Existing classifications are limited in several ways, e.g., in the breadth of data sources, and the generality of proposed categories.
Objective: We aim to characterize code weaknesses in Ethereum smart contracts written in Solidity, and provide an overview of existing classification schemes in relation to this characterization.
Method: We extracted code weaknesses in Ethereum smart contracts from two public coding platforms and two vulnerability databases and categorized them using an open card sorting approach. We devised a classification scheme of smart contract code weaknesses according to their error source and impact. Afterwards, we mapped existing classification schemes to our classification.
Results: The resulting classification consists of 11 categories describing the error source of code weaknesses and 13 categories describing potential impacts. Our findings show that the language-specific coding and the structural data flow categories are the dominant categories, but that the frequency of occurrence differs substantially between the data sources.
Conclusions: Our findings enable researchers to better understand smart contract code weaknesses by defining various dimensions of the problem and supporting our classification with mappings with literature-based classifications and frequency distributions of the defined categories.

2.1 Introduction

To process autonomous tasks, some blockchains execute digital programs called *smart contracts* that become immutable once deployed [35]. Smart contracts are autonomous programs that can (1) customize contracting rules and functions between contractors and (2) facilitate transferring irreversible and traceable digital cryptocurrency transactions [21]. Smart contracts were born and continued to grow with noteworthy achievements in various life fields, including industry, finance, and economics. Smart contracts and blockchains involve a variety of software design and development decisions [46, 25]. For instance, decisions about the logic and development process of a contract, as well as interrelation design decisions such as the dependencies and interactions between contracts and libraries after deployment.

Due to the immaturity of blockchain technology, code weaknesses in smart contracts can have severe consequences and result in substantial financial losses. For instance, the infamous DAO attack in 2016 [31] resulted in stealing around 50 million dollars because of exploiting a re-entrancy vulnerability in the Distributed Autonomous Organizations (DAO) contract[1]. Therefore, understanding code weaknesses in smart contracts is critical to perceive the threats they represent, e.g., to develop predictive models or software engineering tools that can predict or detect threats with a high precision [45].

One suitable way to support understanding in a domain is to structure its knowledge through taxonomies and classification schemes [51, 39]. A classification scheme of code weaknesses in smart contracts is essential, as (i) it can raise awareness among researchers and practitioners about the different categories of code weaknesses in smart contracts, (ii) it helps in the decision-making process and development of smart contract testing and security analysis tools, detection methods, verification methods, and validation approaches, and (iii) it provides a common vocabulary that facilitates communication and focused discussions about contracts' quality in relation to specific architectures or standards, such as ERC token standards [33].

Existing studies attempt to classify code weaknesses in Ethereum smart contracts [4, 17, 11, 54, 23]. While they provide valuable insights into existing security issues in smart contracts, they have the following limitations:

1. **Several studies mix different classification dimensions.** This can lead to classifications where categories overlap, which is problematic if one data point (e.g., a smart contract bug) is assigned to one category only.

[1] https://www.coindesk.com/understanding-dao-hack-journalists

Overlapping categories introduce ambiguity that limit the utility of these taxonomies [39]. For instance, in Zhang et al.'s classification [54] each smart contract bug is classified into one out of nine categories (with several sub-categories each). However, bugs often fit into multiple categories. For example, bugs in the logic sub-category "Denial of Service" can also affect security and would thus fit into both categories. The classification does not clarify how to proceed in this and similar cases. This effectively limits the usability of the classification.

2. **Code weaknesses are classified into broad categories.** Several studies classify code weaknesses into broad categories. However, broad categories are problematic, as the membership in a category does not allow to reason about a code weakness's properties [39]. For example, Dingman et al.'s defect and vulnerability classification [17] contains four categories, including "security". These categories are so broad that they do not provide much information about instances contained therein.

3. **Data sources differ widely.** Several existing smart contract vulnerability classifications rely only on code weaknesses published in academic literature or white literature, e.g., [4] and [3], while [11] used posts on StackExchange, and [54] used a mix of academic literature and GitHub project data. However, biases in data sources, such as representing only part of the overall population of smart contract development, can affect how representative and complete the resulting classifications are. Therefore, existing classifications cannot be compared.

4. **Important data sources are omitted in existing classifications.** To our knowledge, no existing classification scheme or taxonomy for smart contract code weaknesses uses established vulnerability and defect registries such as the Smart Contract Weakness Registry (SWC) [2] and code weaknesses in the Common Vulnerability and Exposure (CVE)[3]. Hence, existing classifications likely suffer from sampling biases.

Therefore, further work is necessary to achieve precise classifications of smart contract code weaknesses. Using fine-grained classes can prevent critical information about smart contract code weaknesses from being missed or lost and improve consistency of analyses among software engineers and experts, especially

[2]https://swcregistry.io/
[3]https://cve.mitre.org/

when evaluating potential security incidents. Hence, this paper aims to characterize code weaknesses in Ethereum smart contracts and present an overview of the related existing classification schemes in relation to the proposed characterization, and devise a unified, generally applicable classification scheme.

The contribution of this study is two-fold:

1. To complement existing studies by classifying smart contract code weaknesses extracted from a variety of important data sources according to the different dimensions presented in existing work.

2. To provide an overview of existing classifications, outlining the different classification dimensions, and mapping existing classifications to our classification scheme using error source and impact as dimensions of the code weaknesses.

We extracted and manually analyzed data related to Ethereum smart contracts written in Solidity from four data sources, i.e., 2065 Stack Overflow posts, 3160 GitHub issues, and 523 and 37 weaknesses from CVE and SWC respectively. Using an open card sorting approach, we devised a classification scheme that uses the error source and impact of a code weakness as dimensions. Furthermore, we mapped existing classifications to this scheme and analyzed the frequency distribution of the defined categories per data source, to reason about potential biases in existing data sources. The resulting classification scheme consists of 11 categories describing the error source, and 13 categories describing potential impacts. We make the data and corresponding analyses publicly available at [29].

Our findings show that most existing classifications use broad categories that are applicable to many code weaknesses and overly generic, such as "security" or "availability". In our classification, language specific coding and structural data flow categories are dominant code weaknesses in Ethereum smart contracts. However, frequency distributions of the error source categories differ widely across data sources. This indicates that classifications based on a single data source should be treated with caution.

The remainder of this paper is organized as follows. Section 2.2 presents the background on Ethereum smart contracts, Solidity and the used data sources. Section 5.7 discusses how existing work relates to our study and what gaps exist. Section 5.3 describes the methodology that we followed. In Section 5.5, we report our findings in terms of the obtained classification and mapping to existing work. We then discuss our findings in Section 2.6 and evaluate the

classification by argument in Section 2.7. Finally, potential threats to validity are reported in Section 5.8 and the paper is concluded in Section 5.9.

2.2 Background

In this section, we discuss the background of our work. Specifically, we discuss definitions used in this paper, the Ethereum environment, existing vulnerability and weakness databases, smart contract security in general, and card sorting techniques.

2.2.1 Definitions

Before we can discuss a classification scheme for smart contract code weaknesses, it is fundamental that we have a sufficiently specific definition of what we are classifying. There have been efforts to formally define concepts such as vulnerability and code weaknesses in SE. However, we will use the formal definition of a vulnerability and weakness that were defined in the Ethereum Improvement Proposals (EIPs)[4] as follows:

- **Vulnerability:** "A weakness or multiple weaknesses which directly or indirectly lead to an undesirable state in a smart contract system" ([19] 1470). A vulnerable contract does not necessarily imply exploited [48]. Moreover, as the contract is a digital agreement between two or more parties, exploiting the contract is not always done by external malicious actors. A venerable contract can also be exploited by one of the contracting parties such as the contract owner who can use vulnerabilities to gain more profits such as in CVE-2018-13783[5]. Any of the contractors can also exploit it, the miners or even the developers who implemented the contract (e.g., CVE-2018-17968 [6]).

- **Weakness:** "a software error or mistake in contract code that in the right conditions can by itself or coupled with other weaknesses lead to a vulnerability" [19]).

What distinguishes smart contract code weaknesses from other software applications is that any smart contract code instruction costs a specific amount

[4]https://eips.ethereum.org/
[5]https://nvd.nist.gov/vuln/detail/CVE-2018-13783
[6]https://nvd.nist.gov/vuln/detail/CVE-2018-17968

of gas (see Section-2.2.4). This means even if the weakness was not exploited, it would result in losing Ether[7] when it is triggered by the contract itself and executed, which makes the contract vulnerable even if it is not exploited.

2.2.2 Ethereum Virtual Machine (EVM)

Ethereum[8] is a globally open decentralized blockchain framework that supports smart contracts, referred to as Ethereum Virtual Machine (EVM). Because EVM hosts and executes smart contracts, it is often referred to as the programmable blockchain. EVM contracts reside on the blockchain in a Turing complete bytecode language; however, they are implemented by developers using high-level languages such as Solidity or Vyper and then compiled to bytecode to be uploaded to the EVM. Users on the EVM can create new contracts, invoke methods in a contract, and transfer Ether. All of the transactions on EVM are recorded publicly and their sequence determines the state of each contract and the balance of each user. In order to ensure the correct execution of smart contracts, EVM relies on a large network of mutually untrusted peers (miners) to process the transactions. EVM also uses the Proof-of-Work (PoW) consensus protocol to ensure that a trustworthy third party (e.g., banks) is not needed to validate transactions, fostering trust among users to build a dependable transaction ledger. EVM gained remarkable popularity among blockchain users as EVM is the first framework that supports smart contracts to manage digital assets and build decentralized applications (DApps) [26].

2.2.3 Ethereum Smart Contracts

A smart contract is a general-purpose digital program that can be deployed and executed on the blockchain. Ethereum smart contract is identified by a unique 160-bit hexadecimal string which is the contract address. It is written in a high-level language, either Solidity or Vyper. In this paper, we focus on Ethereum smart contracts written in Solidity because it is the most popular language in the EVM community and most of the deployed contracts on EVM are written using Solidity [10, 8]. A smart contract can call other accounts, as well as other contracts on the EVM. For example, it can call a function in another contract and send Ether to a user account. In EVM, internal transactions (i.e., calls from within a smart contract) do not create new transactions and are therefore not directly recorded on-chain.

[7] Ethereum corresponding cryptocurrency
[8] https://ethereum.org/en/

2.2.4 Ethereum Gas System

In order to execute a smart contract, a user has to send a transaction (i.e., make a function call) to the target contract and pay a transaction fee that is measured in units of gas, referred to as the gas usage of a transaction. The transaction fee is derived from the contract's computational cost, i.e., the type and the number of executed instructions during runtime. Each executed instruction in the contract consumes an agreed-upon amount of gas. Instructions that need more computational resources cost more gas than instructions that need less computational resources. This helps secure the system against denial-of-service attacks and prevents flooding the network. Hence, the gas system in EVM has two main benefits: (1) it motivates developers to implement efficient applications and smart contracts and (2) it compensates miners who are validating transactions and executing the needed operations for their contributed computing resources. To pay for gas, the transaction fee equals the $gasprice * gascost$. The minimum unit of gas price is Wei (1 Ether = 10^{18} Wei). Therefore, Ether can be thought of as the fuel for operating Ethereum.

2.2.5 Solidity

Solidity[9] is a domain-specific language (DSL) that is used to implement smart contracts on the Ethereum blockchain. It is the most widely used open-source programming language in implementing blockchain and smart contracts. Also, although it was originally designed to be used on Ethereum, it can be used in other blockchain platforms such as Hyperledger and Monax[10]. Solidity is statically typed, which requires specifying the type of all variables in the contract. It does not support any "undefined" or "null" values, and any newly defined variable has a default value based on its type. Smart contracts written in Solidity are organized in terms of subcontracts, interfaces, and libraries. They may contain state variables, functions, function modifiers, events, struct types, and enum types. Also, Solidity contracts can inherit from other contracts, and can call other contracts.

In Solidity there are two kinds of function calls. *Internal function calls*, which do not create an actual EVM call, and *External function calls*, which do. Due to this distinction, Solidity supports four types of visibility for functions and state variables:

[9] https://docs.soliditylang.org/
[10] https://monax.io/

- *External* functions can be called from other contracts using transactions as they are part of the contract interface. They can not be called internally (e.g., *externalFunction()* does not work, while *this.externalFunction()* works). State variables can not be external.

- *Public* functions can be called internally or using messages and they are part of the contract interface. For public state variables, an automatic getter function is generated by the Solidity compiler to avoid high gas cost when returning an entire array.

- *Internal* functions and state variables can only be accessed internally (i.e., from within the current contract or contracts deriving from it).

- *Private* functions and state variables are only visible for the contract they are defined in and not in derived contracts.

State variables can be declared using the keywords *constant* or *immutable*. Immutable variables can be assigned at construction time, while constant variables must be fixed at compile time. For the functions declaration, they can be declared as follows.

- *Pure Functions* promise not to read or modify the state. They can use the *revert()* and *require()* functions to revert potential state changes when an error occurs.

- *View Functions* promise not to modify the state such as writing to state variables.

- *Receive Ether Functions* can exist at most once in a contract. They cannot have any arguments and cannot return any value, and must be declared with external visibility and payable state mutability as in *receive() external payable{ ...}*. These functions are executed on plain Ether transfers (e.g., using *.send()* or *.transfer()*) and based on a call to the contract with empty *calldata*.

- *Fallback Functions* are similar to *Receive Ether* functions, as any contract can have at most one such function. It must have external visibility, and is executed on a call to the contract if none of the other functions match the given function signature or if no data was supplied at all and there is no receive function. The *fallback* function always receives data, but in order to also receive Ether it must be marked as *payable*.

Finally, if any function promises to receive Ether, it has to be declared as *payable*. An example of a Solidity contract is shown in Figure 2.1. It shows a voting contract as explained in Solidity's official documentation [19].

2.2.6 Common Vulnerability and Exposure (CVE) and National Vulnerability Database (NVD)

CVE is a list of publicly disclosed vulnerabilities and exposures that is maintained by MITRE[11]. It feeds into the NVD[12], so both are synchronized at all times. NVD is a comprehensive repository with information about all publicly known software vulnerabilities and includes all public sources of vulnerabilities (e.g., alerts from security focus[13]). NVD also provides more information about the CVE lists' vulnerabilities, such as severity scores and patch availability. It also provides an easy mechanism to search for vulnerabilities using various variables. Both CVE and NVD are maintained by the US Federal Government. An example of a reported smart contract vulnerability in NVD is shown in Table 2.1. It also shows the impact metrics (Common Vulnerability Scoring System - CVSS), vulnerability types (Common Weakness Enumeration - CWE), applicability statements (Common Platform Enumeration - CPE), and other relevant meta-data. Each CVE in the list also has a unique identifier that shows the affected software product, sub-products, and various versions.

2.2.7 Common Weakness Enumeration (CWE)

CWE [15] is a community-developed list of common software security weaknesses. It is considered as a comprehensive online dictionary of weaknesses that have been found in computer software. It also serves as a baseline for weakness identification, mitigation, and prevention efforts. Its primary purpose is to promote the effective use of tools to identify, find, and repair exposures in computer software before the programs are distributed to the public.

[11] https://cve.mitre.org/cve/
[12] http://nvd.nist.gov/
[13] https://www.securityfocus.com/
[15] http://cwe.mitre.org

Table 2.1: An example of a vulnerability in NVD

CVE ID	CVE-2021-3006
NVD Published Date:	01/03/2021
Source:	CVE MITRE
Description	The breed function in the smart contract implementation for Farm in Seal Finance (Seal), an Ethereum token, lacks access control and thus allows price manipulation, as exploited in the wild in December 2020 and January 2021.
CVSS	7.5
Weakness Enumeration	CWE-863
CWE Name	Incorrect Authorization
Hyperlink	Link[14]
Integrity	High
Impact score	3.6

2.2.8 Smart Contract Weakness Classification Registry (SWC)

SWC is an implementation of the weakness classification scheme that is proposed in *Ethereum Improvement Proposals*[16]. It is also aligned with the terms and structures described in the CWE. Each SWC has an identifier (ID), weakness title, relationships, and related code samples list. In the relationships attribute, each SWC is mapped to the parent CWE and CWEs that are related.

2.2.9 Smart Contract Security

Due to the potential severe effects of exploited code weaknesses in smart contracts, security is an important research theme for blockchain platforms such as Ethereum and Hyperledger Fabric [55]. Research on smart contract security follows a number of themes, e.g., principles and patterns of smart contract design [36, 30, 23], secure development [22] and modeling [23], or verification, validation, and auditing [35, 24, 23]. Specifically focusing on vulnerabilities, there has been work studying the security of smart contracts after deployment as well as during execution [22], and suggesting best practices [53].

[16]https://eips.ethereum.org/

In our work, we focus on classifications of smart contract code weaknesses. We will discuss related work in this area more in the next section.

2.3 Related Work

Multiple studies that classify smart contract code weaknesses have been published since the first attack on Ethereum smart contracts in 2016 [16], e.g., [4], [2], [17], and [54]. This section summarizes these studies in relation to our work.

2.3.1 Literature-Based Code Weaknesses Classification

Several studies on Ethereum smart contract weaknesses classify code weaknesses reported in blogs, academic literature, and white papers, i.e., [4, 3, 22, 44, 13, 38, 17].

Atzei et al. [4] propose a classification scheme comprised of three categories to classify code weaknesses in Ethereum smart contracts. In addition to blogs and academic literature, the authors also employ their own practical experience as a resource for their classification. Code weaknesses are classified into language-related issues, blockchain issues, and EVM bytecode issues. The classification is followed by a brief discussion on potential attacks that result in stealing money or causing other damage.

Alharby et al. [3] also use academic literature as the main source of data, classifying them into four main classes: codifying, security, privacy, and performance issues. Additionally, they discuss proposed solutions based on suggestions provided by smart contract analysis tools. The proposed classification suffers from a significant overlap between categories. For example, codifying issues can lead to security and privacy issues, as in the case of the popular re-entrancy code weakness (classified as a security issue in the paper).

Huang et al. [22] report a literature review of smart contract security from a software lifecycle perspective. The authors analyze blockchain features that may result in security issues in smart contracts and summarize popular smart contract code weaknesses based on four development phases, i.e., design, implementation, testing before deployment, and monitoring and analysis. Finally, they classified 10 code weaknesses into three broad categories (i.e., Solidity, blockchain and misunderstanding of common practices). Unfortunately, there is no explanation of how or on what basis these categories were designed.

Dingman et al. [17] study well-known code weaknesses reported in white and gray literature and classify them according to National Institute of Stan-

dards and Technologies Bugs framework (NIST-BF)[17] into security, functional, developmental, and operational code weaknesses . The results show that the majority of code weaknesses fall outside the scope of any category.

Similar to Dingman et al. [17], Samreen et al. [43] survey and map eight popular smart contract code weaknesses in the literature to the NIST-BF. Their results show that only three of the studied eight code weaknesses could be matched with two NIST-BF classes. They also suggest a preventive technique per classified code weakness. Finally, a map between existing analysis tools and the eight code weaknesses are provided in the paper.

Praitheeshan et al. [38] classify 16 smart contract code weaknesses reported in literature based on their internal mechanisms. The authors use three categories, i.e., blockchain, software security issues, and Ethereum and Solidity code weaknesses .

Finally, Chen et al. [13] survey Ethereum system code weaknesses including smart contract code weaknesses reported in literature, classifying them according to two dimensions. First, they group code weaknesses into four-layer groups according to the location, i.e., on the application, data, consensus or network layer. Secondly, they group code weaknesses according to their cause into Ethereum design and implementation, smart contract programming, Solidity language and tool-chain, and human factors. This classification focuses on a few locations that a code weakness might occur at, while omitting others. For instance, the code weakness could be located in the source code of the smart contract itself or in its dependencies.

2.3.2 Repository-Based Code Weaknesses Classification

In addition to classifications that are based on published code weaknesses , two papers attempt classifications based on data extracted from public repositories such as StackExchange, i.e., [11, 54].

Chen et al. [11] collect smart contract code weaknesses from discussions available on Ethereum StackExchange, classifying them based on five high-level aspects according to their consequences, i.e., security, availability, performance, maintainability, and reusability. To evaluate if the selected code weaknesses are harmful, the authors conduct an online survey to collect feedback from practitioners. The proposed categories have the drawback that not all code weaknesses can be clearly placed in a single category, i.e., one code weakness could have various consequences.

[17] https://samate.nist.gov/BF/

Zhang et al. [54] classify smart contracts code weaknesses from both literature and open projects on GitHub. The authors classify extracted code weaknesses into 9 categories based on an extension of IEEE Standard Classification for Software Anomalies[18]. Finally, they propose a four-category classification scheme for the impact of a code weakness, i.e., unwanted function executed, performance, security, and serviceability. Some of the proposed categories are overly general, such as the security category.

2.3.3 Tool-Based Code Weaknesses Detection

As a last category of related work, several publications study existing tools to detect vulnerabilities in smart contracts, and propose classifications based on the tools' capabilities, i.e., [27, 23].

Khan et al. [27] provide an overview of current smart contract code weaknesses and testing tools in their work. In addition, they propose a code weakness taxonomy for smart contract code weaknesses. The proposed taxonomy consists of seven categories, i.e., inter-contractual code weaknesses, contractual code weaknesses, integer bugs, gas-related issues, transnational code weaknesses, deprecated code weaknesses, and randomization code weaknesses. The authors then provide a mapping between the surveyed code weaknesses and the available detection tools. Several of the categories in the proposed taxonomy overlap, e.g., a code weakness caused by an integer bug could lead to gas-related issues.

In a Master thesis, Rameder et al. [23] provides a comprehensive overview of state-of-the-art tools that analyze Ethereum smart contracts with an overview of known code weaknesses and the available classification schemes in the literature. The studied code weaknesses are classified into 10 novel categories, i.e., malicious environment, environment dependency/blockchain, exception & error handling disorders, denial of service, gas related issues, authentication, arithmetic bugs, bad coding quality, environment configuration, and deprecated code weaknesses .

2.3.4 Summary and Research Gap

In summary, various attempts to classify smart contract code weaknesses have been published, both on reported code weaknesses and by mining software repositories. Another line of research focuses on studying smart contract code weakness detection tools, classifying what code weaknesses they are able to detect. Overall, the proposed classifications are valuable to study smart contract

[18]https://standards.ieee.org/standard/1044-2009.html

code weaknesses from different dimensions, e.g., considering where in the network code weaknesses occur, or what the impacts of the code weaknesses are. However, existing classifications suffer from a number of flaws. First, several categorizations rely exclusively on code weaknesses reported in literature and might, therefore, provide a skewed image, e.g., [4]. Secondly, we observe that many classifications mix different concerns or dimensions, such as consequences of exploiting a code weakness and the source of error in [23]. Third, existing categorizations propose general categories that do not allow for detailed reasoning, such as the privacy and security categories in [3] and [54]. Finally, they provide only a limited view on smart contract code weaknesses due to focusing on a single dimension, such as the consequences in [13]. Therefore, there is a need to provide a reference taxonomy that includes several dimensions such as root cause, impact, or scope [32]. The aim of this paper is to arrive at such a classification by integrating existing work and complementing it with additional data from software repositories and well-known sources such as the CVE and SWC registries.

2.4 Research Method

This section presents the method we followed in this paper. Figure 3.1 shows an overview of the study method. Initially, we selected four data sources (Steps 1-4), i.e., Stack Overflow, GitHub, CVE, and CWE. We then collected data from the respective sources, cleaned and pre-processed it (Step 5). This was followed by manual labeling using open card sorting (Step 6). In Step 7, the resulting smart contract code weakness and impact classifications were mapped to literature-based classifications. We then defined critical code weaknesses and explained them (Step 8). As a final step, we analyzed the cleaned data quantitatively. A detailed description of each step follows in the remainder of this section.

2.4.1 Study Setup

In this study, we aim to answer the following research questions (RQs):

- *RQ1. What categories of code weaknesses appear in smart contracts?*

  Goal: To comprehensively categorize the code weaknesses that appear in Ethereum smart contracts. To study different dimensions of the problem and to map literature-based classifications on Ethereum smart contract

code weaknesses and unify them in one thorough classification. A thorough classification scheme will make it possible to collect statistics on frequency, trends, and code weaknesses, as well as evaluate countermeasures.

- *RQ2. How do frequency distributions of smart contract code weakness categories compare across data sources?*

  Goal: To investigate if all data sources have the same frequency distributions of code weaknesses. If so, then we can rank code weakness categories from the most common to the least common. If not, we can reason about the skew or bias when different sources are used. Further research can find solutions for the most common code weaknesses. More effort could be put to address the prevalent category before deploying the contract to the blockchain.

- *RQ3. What impact do the different categories of smart contract code weaknesses have?*

  Goal: To investigate the impacts of smart contract code weakness on the product and the business. To define various dimensions of impacts classifications and to map literature-based impact classifications and propose a thorough impact classification of smart contract code weaknesses. More effort can be put to code weaknesses with critical impacts.

Data Sources

To answer the proposed research questions, we analyzed and studied smart contract code weaknesses from four primary sources. Two of these sources are widely used by developers (i.e., Github and StackOverflow), and two are very well-known publicly accessible sources (i.e., SWC and CVE) for reporting code weaknesses in Ethereum smart contracts and other software systems. Table 4.2 shows the final number of data records during and after data pre-processing. In all data sources, we extracted only records that were marked as vulnerable. That is, we did not judge whether or not smart contract code was indeed vulnerable. To assess the impact, we only used Stack Overflow, since posts included information on what went wrong in a smart contract. In GitHub, SWC and CVE, impact information was not readily available.

Data-source 1: We opted for extracting data from Stack Overflow, as it has successfully been leveraged in existing work on smart contracts [6, 7, 11], and in general Software Engineering research [9, 37, 11, 12, 1].

We used Stack Overflow posts to study code weaknesses in smart contracts. To do so, we extracted Q&A posts tagged with "smart contract", "Solidity", and "Ethereum", posted between January 2015 and April 2021. We discarded posts with the "Ethereum" tag, but without the "Solidity" or "smart contract" tags. To retrieve the related information for each post, we used Scrapy[19], an open source Python library that facilitates Web crawling. For each of the 2065 extracted posts, we extracted the post title, URL with the description of the code weakness, related tags, post time and accepted answer time.

Data-source 2: We used GitHub as the second data source for our study, as it is the most popular social coding platform [15]. Moreover, many studies on Ethereum smart contracts and smart contract analysis tools have been published reporting findings based on data published in GitHub open-source projects (e.g., [16]). We studied code weaknesses reported in open source projects that have Ethereum smart contracts written in Solidity. We used the keywords "smart contract", "Solidity", and "Ethereum" to search for these projects. We also collected the code weaknesses found in the closed reports (i.e., the closed issues that have the keyword "fix") in GitHub.

We also studied the official Ethereum GitHub repository. Figure 2.3 shows an example of a reported smart contract code weakness in GitHub.

Data-source 3: We used the NVD search interface to collect and extract all reported CVEs on smart contracts and their CVSS until April 2021. We searched with "smart contract", "Solidity", and "Ethereum". Then, we manually extracted the reported CVEs that are related to smart contracts.

Data-source 4: We extracted all the reported code weaknesses in SWC until April 2021. For each code weakness, we extracted the ID, title, relationships, and test cases.

The collected data records were stored in Excel sheets (one for each data source) for later pre-processing and cleaning.

Data Cleaning and Pre-Processing

After the initial extraction, we applied several filtering steps to obtain a clean dataset. First, we removed posts with duplicate titles or marked as *[duplicate]* keyword in Stack Overflow, and we manually inspected the title and the body of the question and decided if the post actually discussed smart contracts in Ethereum/Solidity. Finally, we removed vague, ambiguous, incomplete, and overly broad posts. As an indication for such posts, we considered the amount

[19] `https://scrapy.org`

of negative votes and/or negative feedback. The threshold of the negative votes was set to two negative votes or two negative comments. Finally, we extracted the code of the smart contract in each post for further analysis.

In order to pre-process the collected GitHub data, we removed the duplicates based on the description URL of the code weakness. If two records in the database with the same code weakness description URL and type, then we considered these two records as duplicate code weakness and removed one of them. For further analysis, we also extracted the code of the smart contract that contained the code weakness. We also manually double-checked data collected from both the NVD database and SWC registry for duplicates. After this stage, we had a clean dataset with records from the four data sources.

Table 4.2 shows a summary of the collected dataset after applying the aforementioned cleaning and pre-processing steps.

Table 2.2: Summary of sampled code weaknesses in selected data sources

Data Source	Stack Overflow	GitHub	CVE	SWC
# of collected data	2065	3160	523	37
# of data records after preprocessing steps	1490	1160	523	37
Final # of Code weaknesses	765	818	523	37

2.4.2 Data Analysis and Classification Categories

To analyze and label the cleaned dataset, we manually inspected each record and read the description of each code weakness. Then, to define the categories of code weaknesses in smart contracts, two experts (i.e., a doctoral student[20] with a software engineering background focusing on smart contract security, and a cybersecurity expert with experience in cryptocurrencies, smart contracts, and security incidents) together used open card sorting [48] to propose a classification scheme based on the cleaned data. We used an open card sorting approach, since code weakness categories in the selected data were unknown and could potentially differ from existing classifications. Furthermore, open card sorting

[20]The first author of this paper.

allowed us to define categories that are more specific compared to existing ones, and potentially more useful. Finally, the approach ensured the classification is grounded in real-world smart contracts and developer discussions. Open card sorting is appropriate in this context, as designing the different code weakness categories is a creative, qualitative task. Disagreements between the two experts were used to discuss potential differences and come up with a categorization of higher quality. Data sources were not weighted anyhow, as all records were taken into account.

Card Sorting for Taxonomy Construction

Card sorting, a prevalent user-centered design technique, involves prompting participants to categorize a set of cards into meaningful groups, thereby revealing their implicit item grouping tendencies [54]. Widely utilized in the construction of taxonomies, card sorting provides valuable insights into the process of information categorization [48]. It allows researchers to gain a deeper understanding of the inherent structure and relationships within a given set of items. Each card or item can be an object, a picture, an attribute name [28, 41]. There are different types of card sorting techniques. In open card sorting, participants have the freedom to generate their own categories and allocate items without predetermined labels or constraints [42]. Conversely, closed card sorting provides participants with predefined categories into which they sort the items [42]. Hybrid card sorting combines elements of both open and closed sorting, enabling participants to create new categories while also offering a predefined set of categories as a reference [32]. The selection of a specific card sorting technique depends on the research objectives, contextual factors, and available resources.

Open card sorting settings participants, and materials. The open card sorting exercise was conducted over approximately 6 months in total, with the two experts involved throughout. The experts were selected based on availability and their experience in relevant areas. As materials, we used a list of the selected data sources and an introduction to the open card sorting exercise with the goal of the exercise. We also provided a description of what information each card should contain with a brief description of the information extracted from each data source as mentioned before. The two experts then discussed the sorting process.

For each record in the cleaned dataset, we created a card containing the information of the code weakness as collected from the original data source and a unique ID. A summary of the information included in the cards for each data source is shown in Table 2.3. All cards were stored on two separate computers, as to avoid the experts influencing each other.

Table 2.3: Overview of the information included in the card for each data source

Data Source	Card information
Stack Overflow	Tag, title, description, expert 1 label, expert 2 label, and agreement.
GitHub	ID, description URL, expert 1 label, expert 2 label, and agreement.
CVE	CVEName, description, CWE, expert 1 label, expert 2 label, and agreement.
SWCs	Title, description, CWE, expert 1 label, expert 2 label, and agreement.

Open Card Sorting Procedure. First, the two experts labeled a random 10% of the data, and measured inter-rater agreement after independent labeling using Cohen's Kappa coefficient. The Kappa value between the two experts was 0.60, showing moderate agreement [59].

After this session, the two experts discussed the categories in a clarification session, where categories deemed overly broad were discarded. Some categories were merged, and the two experts agreed on the naming of the resulted fine-grained classes. An example of two merged categories are "access control" and "authentication", where the experts decided to use "authentication and authorization" as a common category. The categories were generally labeled based on the causes mentioned in the descriptions in each card. That is, we analyzed the question and the answer information on Stack Overflow, code weakness abstract and details on GitHub, CVE description in the NVD database, and the SWC's description.

Disagreements among the experts were resolved through discussions. Labeling took two to three hours every weekday. For instance, the experts disagreed whether or not "structural data flow" should be included in the category named "logic". Ultimately, both categories were kept, as the causes for the weaknesses differed in the data. Common sources of disagreement were cases were causes of a weakness were not stated clearly in the card. In such cases, the experts consulted the Solidity code connected to the reported weakness and discussed

the cause until reaching an agreement. After completing the first 10% of the data, the existing categories were used as base categories for the remainder of the data.

The two experts then followed the same procedure, with one open card sorting session per working day. Clarification discussions were also repeated after 20% and 40% of the posts were labeled (with Kappa values of 0.75 and 0.82 respectively). Similarly to the first classification session, these sessions were used to clarify major disagreements. Finally, calculations of the Kappa coefficient at 60%, 80%, and 100% resulted in values of > 0.90, indicating perfect agreement.

The same approach was followed to label the impact of each code weakness in the cleaned dataset. The Kappa value was also > 0.90 in the final discussion of the impact labeling.

2.4.3 Unifying Classification Schemes

To unify existing code weakness classification schemes in the literature, we gathered all the categories proposed in the literature into an Excel sheet. The first and the second authors then discussed each category in relation to the categories proposed by us. We subsequently defined three dimensions of the problem (i.e., the code weakness's source of error; the location of the code weakness in the network level; the behavior and consequences arising from an exploit of the code weakness). Afterwards, we mapped existing classifications to the defined dimensions, as illustrated in Figure 2.4. We name these code weakness dimensions *V-D*. The error source dimension (V-D1) describes the main cause that, when triggered, can result in executing the code weakness, such as the logic of the contract and the data initialization. V-D2 is the network dimension, which indicates at which layer on the network the code weakness occurs. Ethereum network consists from several layers; (1) application layer (i.e., the code weakness can be in the user account), (2) blockchain data layer (i.e., the weakness can be in the blockchain data structure), or (3) consensus layer (i.e., the weakness can be at the consensus protocols), etc[13]. Finally, V-D3 describes the resulting behavior and consequences of the code weakness, which means the result of executing the code weakness.

We followed the same approach to devise a thorough classification scheme for the impacts of smart contract code weaknesses. We defined two dimensions (i.e., impact on the software product, and impact on business factors) as shown in Figure 2.5. We name these impact dimensions *I-D*. I-D1 is the impact of the code weakness on the software product itself (i.e. the smart contract as a digital program) and its resulting behavior. For instance, software engineers can de-

scribe the poor performance of a contract. As such, I-D1 considers the behavior of smart contracts from a software engineering perspective. I-D2 describes the impact of the code weakness on business factors and the contract owners, e.g., losing money or important information of the business owners.

2.5 Results

In this section, we present the results of our study, and answer the proposed RQs. Our objective is to unify the existing classification schemes and define the causes, impacts, and recurrences of smart contract code weaknesses. Moreover, we propose thorough classification schemes for both the impacts and categories of smart contract weaknesses.

> Smart contract code weaknesses are found in more than half (i.e., 66.7%) of the cleaned data from all the four data sources (Stack Overflow, GitHub, CVE and SWC).

2.5.1 What Categories of Code Weaknesses Appear in Smart Contracts? (RQ1)

To answer RQ1, we followed the analysis approach in Section 2.4.2, then mapped the result of our classification to other classification schemes as explained in Section 2.4.3. We classified the 2143 extracted weaknesses into 47 unique smart contract weaknesses, grouped into 11 categories. Within our classification scheme, we mapped the existing literature classification schemes. We were able to define 47 critical code weaknesses from the data we collected. In this work, 21 novel code weaknesses are introduced to the body of knowledge.

Table 2.4 briefly summarizes our proposed categories of smart contract code weaknesses and their definitions. The *Category Name* column lists the name assigned to each category, while the *Description* column defines the nature of the code weakness. The *Short* column presents abbreviations derived from the *Category Name* for ease of reference in the following sections and charts. For example, *Structural Data Flow* is abbreviated as *(SV-DF)*, where *"SV"* refers to *"Structural Weakness"* and *"DF"* to *"Data Flow"*.

Table 2.5 maps literature-based classification schemes of smart contract code weaknesses with our own. The categories proposed in the literature are listed in the rows, while our categories are listed in the columns of the table. The

table covers all the three dimensions of *V-D* discussed in Section 2.4.3. As can be seen, some broad categories listed in literature essentially cover all of our classification categories.

> Most literature-based classification schemes for smart contract code weaknesses include broad categories, such as security and availability.

The following subsections present the key findings related to the defined categories. For each category, we define and explain the most critical code weaknesses as agreed by the two raters. We give examples for some weaknesses, which are directly taken from our dataset. For the full list of code weakness definitions, we refer to our published dataset [29].

Language Specific Coding Weaknesses

In this category, smart contract code weaknesses result from language-based errors not captured by the compiler. The source of error of this category can be in the language pre-defined functions, events, libraries, and/or language standards.

We define 14 Language Specific Coding weaknesses that can result in undesirable state in a smart contract or can be used by attackers in their favor. Next, we show a sample of these weaknesses.

Insufficient compiler version or pragma version — CV#1. A so-called version *pragma* should be included in the source code of smart contract to reject compiling the contract using incompatible compiler versions. When using a version *pragma* in the contract which is later than the selected compiler, this may introduce incompatible changes and lead to code weaknesses in compiled smart contract code. Moreover, future compiler versions may handle language constructs in a way that introduces unclear changes affecting the behavior of the contract as shown in Listing 2.1.

```
1 pragma solidity ^0.6.3; // weakness
2 pragma solidity 0.6.3;
```

Listing 2.1: Version pragma

Fallback function not payable — CV#2. Smart contracts written in Solidity versions 0.6.0+ should have the fallback function split up into a *receive()* and a *fallback()* function (i.e., a new fallback function that is defined using the

Table 2.4: Classification scheme of code weaknesses in smart contracts

Category Name	Description	Short
Language Specific Coding	Syntax mistakes in implementing Solidity contracts that are not captured by the Solidity compiler can introduce unexpected behavior or damage the contract.	CV
Data Initialization and Validation	The input data type to the contract or the fields' data type in the contract are not initialized correctly. Also, includes the data passed from/sent to other contracts. **Predictable resources** is a subcategory of data initialization and validation. Weaknesses resulting from using expected values in state variables or functions. These code weaknesses can allow malicious minors to take advantage of the code weaknesses and control the contract.	DV DV-PR
Structural Sequence & Control	Problem with the structure of the contract control flow. Specifically, a result of incorrect control flow structure such as require, if-else conditions, assert, and loop structures.	SV-SC
Structural Data Flow	Problems with the structure of the contract data flow. The main difference to *Data* is that *Data* originates in the data fields and input parameters to the subcontracts or the contract methods. Instead, in this category, changing data fields in a wrong way during and after the execution of the contract leads to issues.	SV-DF
Logic	Inconsistency between the intention of the programmer and the coded contract, and not one of the other categories.	LV
Timing & Optimization	Performance and timing issues that can affect execution time and results in abnormal responsiveness under a normal workload.	TV
Compatibility	Required software and packages are not compatible with the available resources (e.g., operating system, CPU architecture).	CoV
Deployment & Configurations	Weaknesses during deployment of implemented contracts on the Ethereum blockchain.	DL
Authentication & Authorization	Code weaknesses allow malicious actors to take control over the contract.	SV
Dependency & Upgradability	Upgrades in a smart contract breaking the dependencies in the new contract.	UV
Interface	Code weaknesses in the interface of smart contracts, e.g., when the contract is functioning correctly, but the interface is showing a wrong output that contradicts the contract's execution logs and transaction logs.	IB

Table 2.5: Mapping literature-based classifications to V-D. $\subset^*$ indicates that the category in the V-D classification is a subset of the corresponding category in the literature marked by *. $^*\subset$ means the category in the literature is a subset of a proposed category in V-D. Finally, = means the categories are identical.

Category	CV	DV	SV-SC	LV	TV	CoV	DL	SV-DF	UV	IB
Codifying [3]	$\subset^*$			$\subset^*$						
Data* [54]	$^*\subset$	$^*\subset$								
Description* [54]	$^*\subset$									
Interaction* [54]	$^*\subset$	$^*\subset$		$^*\subset$						
Interface* [54]										=
Logic* [54]				=						
Standard* [54]	$^*\subset$									
Authentication* [23]								=		
Arithmetic* [23]				=						
Bad Coding Quality* [23]	=									
Environment Configuration* [23]							=			
Deprecated* [23]	$^*\subset$									
Solidity* [4]	$^*\subset$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$		$\subset^*$	$\subset^*$	$\subset^*$	
EVM* [4]						$\subset^*$				
Blockchain* [4]							$\subset^*$			
Security* [3]	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$
Privacy* [3]	$\subset^*$	$\subset^*$		$\subset^*$						
Performance* [3]					=					
Availability* [11]	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$
Maintainability* [11]	$\subset^*$		$\subset^*$	$\subset^*$		$\subset^*$			$\subset^*$	
Reusability* [11]	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$			$\subset^*$	$\subset^*$		$\subset^*$

fallback keyword and a *receive* ether function defined using the *receive* keyword). If present, the *receive* function is called whenever no parameters are provided in the call. The *receive* function is implicitly payable. The new *fallback* function is called when no other function matches. However, if *fallback()* is not payable and *receive()* does not exist, transactions not matching any other function which send ether will revert and result in an undesirable state in the contract.

Fallback function does not exist — CV#3. In addition to CV#2, when sending Ether from one contract to another contract without calling any of the receiving contract's functions, sending the Ether will fail if the receiver contract has no fallback function. Thus, a payable fallback function should be added to the receiver before deployment. Otherwise, there is no way to receive the Ether unless the sender has previous knowledge of the exact functions of the receiving contract, which is not usually the case.

Violating splitting Fallback function — CV#4. Smart contracts written in Solidity versions 0.6.0+ should have the fallback function split up to *receive()* and *fallback()*. If present, the *receive* function is called whenever the call data is empty. The *receive* function is implicitly payable. The new *fallback* function is called when no other function matches. The fallback function in Listing 2.2 can be payable or not, however, if it is not payable then transactions not matching any other function which send value will revert.

```
1 contract payment{
2   mapping(address => uint) _balance;
3   fallback() payable external {
4     _balance[msg.sender] += msg.value;
5   }
6 }
```

Listing 2.2: Fallback function

Violating modifier definition — CV#5. Solidity provides modifiers that are used to change the behavior of functions in a declarative way. They can be used to enforce pre/post-conditions on the execution of a function. The _ operator should be used in defining a modifier, and starting from Solidity version 0.4.0+ a semicolon should be added after the _ operator. The operator represents the actual code of the function that is being modified. Thus, the code for the function being modified is inserted where the _ is placed in the modifier. Missing the _ operator might generate unwanted results. For example, in Listing 2.3, line 8, every time transferOwnership is invoked, the onlyOwner modifier will get into play first. If the owner invokes it, then the control flow

will reach the _ operator, so the transferOwnership statements will be executed. Otherwise, the execution will just throw, revert, and exit.

```
contract owned {
  address public owner;
  function owned() {
    owner = msg.sender;}
  modifier onlyOwner {
    if (msg.sender != owner) throw;
    _;  }
  function transferOwnership(address newOwner) onlyOwner {
    owner = newOwner; }}
```

Listing 2.3: Violating *modifier* definition

Manipulated language standard — CV#6. Ethereum has adopted many standards to guarantee the composability of smart contracts and Dapps. Those standards are in Ethereum's official EIPs and include token [21] standards. For example, ERC-20 is a token technical standard that allows developers to implement tokens of cryptocurrencies. It contains nine unique functions and two events to guarantee the possibility of exchanging tokens based on ERC-20 with other ERC-20 tokens. Any modification on the function name, parameter types, and the return value in the standard might change its functionality and leave the developer believing it is the same as ERC-20. The implementation of ERC-20 in any contract shall strictly be the same as in the standard template.

Violating call-stack depth limit — CV#7. In Ethereum, the call-stack has a hard limit of 1024 frames. Each time the contract calls an external contract, the call-stack depth of transaction increases by one. Thus, when the number of calls exceeds the limit of the call-stack, an exception is thrown and the call is aborted by Solidity. Moreover, Solidity does not support exceptions in low-level external calls. Therefore, a malicious actor can recursively call a contract 1023 times, then call a victim contract to reach the call-stack depth limit. This will fail any subsequent call made by the victim contract without the victim contract owner being aware of the attack. Recently, EIP 150[22] makes it impossible to reach stack depths of 1024, effectively eliminating call depth attacks

Insufficient Address split — CV#8. Starting from Solidity 0.5.0+, the address should be split to address and address payable, where only ad-

[21] An Ethereum token can represent anything, including lottery tickets, financial assets, a fiat currency like USD, an ounce of gold, etc.

[22] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-150.md

dress payable provides the transfer function. Otherwise, sending tokens to "unpayable" addresses will be reverted. Moreover, there is no way to convert an address to address payable.

Mixing *pure* and *view* — CV#9. In Solidity, *pure* and *view* are function modifiers that describe how the logic in that function will interact with the contract's state. Functions that are declared *view* promise not to modify the state, while functions that are declared *pure* promise not to read or write the state.By using no specifier, the state can be read as well as modified. Developers can mix these two modifiers by replacing *view* with *pure* or missing any of these modifiers, resulting in unexpected state changes or incorrect reads from the state .

Using of *balance* as attribute to the contract — CV#10. One of the features of Solidity is that contracts inherit all members from *Address*, meaning that the keyword *this* is the pointer to the current instance of the type derived from Address. In other words, if the developer wants to access members of the address type (e.g. the balance) of the current contract instance, then the developer should use *this* and should use *balance* as an attribute of the address type, not the contract as shown in Listing 2.4. We noticed a confusion in using balance and other address attributes as if they are attributes of the contract.

```
function getSummary() public view returns(
  uint, uint, uint, uint, address
){
  return (
    minimumContribution,
     this.balance, //incorrect
    address(this).balance,// correct
    requests.length,
    approversCount,
    manager
  );
}
```

Listing 2.4: Using of *balance* as attribute to the contract

Unsafe delegatecall (code injection)— CV#11. A special variant of a message call in Solidity is *delegatecall*. With this feature, the contract can be executed in the callee's context, while msg.sender and msg.value remain unchanged. Consequently, it allows an external contract to modify the storage of the calling contract. This can be exploited by a malicious caller to manipulate the caller's contract state variables and take full control over the balance.

Variable shadowing — CV#12. Solidity supports ambiguous naming

when inheritance is used. For instance, contract Alpha with a variable V could inherit contract Beta that also has a state variable V defined. Consequently, there would be two versions of V, one accessed from contract Alpha and the other from contract Beta. In complex contract systems, this condition might go undetected and ultimately cause security issues. Also, this can also occur at the contract level (e.g., a contract with more than one definition at the contract and function level).

Deprecated Solidity code — CV#13. As Solidity evolves, several of its functions and operators are deprecated. Making use of them leads to poor code quality. It is strongly discouraged to use deprecated Solidity language code with new major versions of the Solidity compiler, since it can cause unwanted behavior.

Experimental Language Features — CV#14. Similar to CV #13, it is strongly discouraged to use experimental Solidity language features since it can cause undesired behavior.

Data Weaknesses

Most of the data weaknesses result from the use of wrong or insufficient data types, or passing wrong data formats to arguments without knowing the exact required type. Moreover, organizing the memory and storage in Solidity is the responsibility of programmers, which many developers are not used to do.

Violating explicit data location — DV#1. For Solidity versions 0.5.0+, an explicit data location for all variables of type struct, array, or mapping is mandatory. This also applies to function parameters and return variables. For instance, *calldata* is a special data location that contains the function arguments, which is only available for external function call parameters. If *calldata* is not included in the initialization, it results in unexpected values as shown in Listing 2.5.

```
1  contract StructExample {
2
3    struct SomeStruct {
4      int someNumber;
5      string someString;
6    }
7    SomeStruct[] someStructs;
8
9    function addSomeStruct() {
10     SomeStruct storage someStruct = SomeStruct(123, "test");// insufficient use
11     SomeStruct memory someStruct = SomeStruct(123, "test");// correct
```

```
12      someStructs.push(someStruct);
13    }
14  }
```

Listing 2.5: Using of *torage* instead of *memory*

Using of *storage* instead of *memory* — DV#2. In addition to *call-data*, Solidity provides two more reference types to comprise structs, arrays, and mappings called *storage* and *memory*. The Solidity contract can use any amount of memory (based on the amount of Ether that the contract owns and can pay for) during execution. However, when execution stops, the entire content of the memory is wiped, and the next execution will start fresh. The *storage* is persisted into the blockchain itself, so the next time the contract executes, it has access to all the data it previously stored in its storage area. Confusing storage and memory can result in data loss.

Violating array indexing — DV#3. Developers are making numerous mistakes when initializing and accessing arrays in Solidity. Most of the time, discovering these violations is not easy, especially if there is no syntax error or an error that can be detected by the compiler. This can result in returning incorrect values. A clear violation of array indexing is shown in Listing 2.6, line 9, where the developer is trying to access a single element in a 3-dimensional array, but only provides two sets of square brackets. Therefore, the developer is returning an array instead of a single Object.

```
1  contract Game {
2    struct User{
3      address owner; }
4  User[][10][10] public gameBoard;
5  User memory mover = gameBoard[_fromX][_fromY][0];
6     function addUser (uint _x, uint _y) public {
7          gameBoard[_x][_y].push(User(msg.sender, 10, 5, 5, 5, 5));}
8          function moveUser (uint _fromX, uint _fromY) public {
9          User memory mover = gameBoard[_fromX][_fromY]; //incorrect access
10         if (mover.owner != msg.sender)return;}}
```

Listing 2.6: Violating arrays indexing

Hard-coded address — DV#4. An existing bad practice is the use of hard-coded addresses in smart contract code, as shown in Listing 2.7. Any incorrect or missing digit in the address may result in losing Ether, in case Ether is sent to that wrong address, or in unexpected outcomes. This code weakness is also known as "Transfer to orphan address" [4].

```
address recieveraddress ;
function initializeAddress1 () {
  recieveraddress = 0x98081c...8e5ace; //hardcoded address}
```

Listing 2.7: Hardcoded address

Improper data validation — DV#5. It is necessary to validate input from untrusted sources, such as external libraries or contracts before integrating it into any contract logic.

Unintentional Write to arbitrary storage location — DV#6. Because Solidity storage is not dynamically allocated, it can lead to unpredictable behavior, unauthorized data access, and other code weaknesses, especially if the data location of data types like structs, mappings, and arrays is not clarified and allowed to overwrite entries of other data structures.

Predictable Data Values and Resources

We encountered several issues in Solidity smart contracts that relate to values that can be guessed even though they are intended to serve as an element of randomness.

Timestamp dependency — DV-PR#7. To keep contracts safe from malicious actors, developers should avoid using the block variables as a source of randomness or as part of triggering conditions for executing significant operations in their contracts, such as transferring Ether. When submitting blocks, miners determine the value of block variables such as block.timestamp, block.coinbase, and block.difficulty. Thus, these values can affect the contract's outcome and can be used to benefit the attacker. For example, Listing 2.8 shows an insecure lottery contract in which block.timestamp is used as a source of entropy.

```
function setWinner() public {

  bytes32 hash = keccak256(abi.encode(block.timestamp));
  bytes4[2] memory x0 = [bytes4(0), 0];
  assembly {
    mstore(x0, hash)
    mstore(add(x0, 4), hash)
  }
```

Listing 2.8: Insecure lottery contract using block variables

Blockhash dependency — DV-PR#8. Using blockhash has the same risks as block.timestamp in DV-PR #7, especially when used in critical operations such as Ether transfer. It can lead to serious attacks as malicious miners can tamper with the blockhash and take full control over the contract.

Bad random number generation— DV-PR#9. Using random numbers is not avoidable in some smart contracts, e.g., games or lotteries. It is important that the randomness is not based on global blockchain variables, as that leaves the contract open to manipulation by malicious miners similar to DV-PR#7 and DV-PR#8.

Sequence and Control Weaknesses

This category of code weaknesses is corresponding to incorrect control structure and loop control statements. These can be exploited and help the attacker to steal money in the contract. Moreover, they can also result in losing all the money in the contract without even being attacked, just because of code weaknesses in these structures.

Using assert instead of require — SV-SC#1. The *assert* statement should be only used for conditions that indicate you have an internal code weakness in the contract code. The *require* statement should be used to check valid conditions (e.g., state variables conditions are met, validate input, and validate return value from external contracts). A valid code with correct functions should never fail *assert* conditions. Otherwise, there is a code weakness in the contract and something unexpected has happened. In smart contracts, *assert* can consume all the gas in the contract, as shown in Listing 2.9. If the example is tested with run(8), the function runs successfully and 1860 gas will be consumed based on the cost of the function and the loop iterations[23]. If it is tested with run(15), then the function passes assert, fails at require and the first loop only will be executed and consume 1049 gas. Finally, testing the same example with run(25) causes the function to fail the assert statement. The execution continues and thus iterates 25 times through the loop, resulting in a very high cost of gas.

```
1 contract Test {
2     function run(uint8 i) public pure {
3         uint8 total = 0;
4         for (uint8 j = 0; j < 10; j++)
5           total += j;
```

[23]The actual gas costs are stated in the Solidity documentation and depend on numerous factors, such as the executed functions and the used data types.

```
6         assert (i < 20);
7         require (i < 10);
8         for (j = 0; j < 10; j++)
9           total += j;
10    }
```

Listing 2.9: Using assert instead of require

Data Flow Weaknesses

These are code weaknesses in the data flow of smart contracts, so that data fields are changing unexpectedly or incorrectly. We defined two code weaknesses that belong to this category.

Updating storage in fallback functions— SV-DF#1. Upon receiving Ether without a function being called, either the receive Ether or the fallback function is executed. If the contract does neither have a receive nor a fallback function, the Ether will be rejected by throwing an exception. During the execution of one of these functions, the contract can only rely on the passed gas (i.e., 2300 gas) being available to it at that time. This stipend is not enough to modify storage. However, we found that developers sometimes are updating state variables, trying to write to the storage in the fallback functions. Updating the variables with such a gas limit will fail. If the data flow of the contract depends on the failed state variables, this results in an incorrect data flow and in unexpected outcomes.

Arithmetic operation/overflow and underflow — SV-DF#2. An overflow can happen as a result of an arithmetic operation or calculation that falls outside the range of a Solidity data type, resulting in unwanted behavior or unauthorized manipulation of the contract balance. Underflow happens when an arithmetic operation reaches the minimum size of a type. This is a data flow code weakness, as the code of the contract does not perform correct validation on the numeric inputs and the calculations. In addition, the Solidity compiler does not enforce detecting integer overflow/underflow. An example is shown in Listing 2.10, where the computation of *mask* overflows at x >= 248.

```
1 uint256 public MAXUINT256 = 2*256 - 1;
2 for (uint256 x = 0; x < 255; x++) {
3    var mask = MAXUINT256 * (2 ** x);
4    var key = signature & bytes32(mask);}
```

Listing 2.10: Arithmetic Operation/calculation overflow

Logic Weaknesses

This category reflects inconsistencies with the contract and the programmer's intention, which is usually mentioned in the question information. These issues relate to a number of reasons, e.g., vague developer intentions, misunderstanding of language components, incorrect usage of Math, and incorrect gas predictions.

Greedy contract— LV#1. This code weakness occurs when implementing a contract logic that is only locking Ether balance all the time because of its inability to access the external library contract to transfer Ether. For instance, the contract logic may only accept transferring money based on a specific value in the code, which happens to be unreachable due to incorrect logic. In this case, the Ether will be locked in the deployed contract forever. In the example of Listing 2.11, the function *refundMoney()*, line 8, does not decrease the *weiRaised* value, meaning that once starting a refund, the developer can no longer use *forwardAllRaisedFunds()* to drain the contract. This code weakness would be triggered even in the regular course of action and Ether in this contract is stuck. It can receive any funds but the received funds can never be retrieved.

```
1  contract SwordCrowdsale is Ownable {
2      //amount of raised money in wei
3      uint256 public weiRaised;
4      bool public isSoftCapHit = false;
5      //send ether to the fund collection wallet
6  function forwardAllRaisedFunds() internal {
7      wallet.transfer(weiRaised);}
8  function refundMoney(address _address) onlyOwner public {
9      uint amount = contributorList[_address].contributionAmount;
10     if (amount > 0 && _address.send(amount)) {
11     //user got money back
12     contributorList[_address].contributionAmount =  0;
13     contributorList[_address].tokensIssued =  0;}
14     }
```

Listing 2.11: Greedy Contract

Transaction order dependency — LV#2. The code weakness arises when a contract's logic is dependent on the order in which transactions are executed and processed in a block. It is a type of race condition inherent to Blockchains. By manipulating the order of transaction processing, malicious miners can take advantage of the contract and benefit from it. Therefore, the logic of the contract should not rely on the transaction order.

Call to the unknown — LV#3. This code weakness arises when a func-

tion unexpectedly invokes the fallback function of the recipient. Consequently, malicious code can be introduced. For example, the unknown call could trigger the recipient's fallback function, allowing malicious code to execute.Also, this can be done via direct call, delegatcall, send, or only call functions. In the MultiSig Wallet Attack[24], an attacker exploited this code weakness to steal 30M USD from the Parity Wallet. Another example is shown in Listing 2.12. In which, pong function uses a direct call to invoke Alice's ping. However, if the interface of contract Alice by mistake was mistyped by declaring the parameter as *int* instead of *unit* and Alice has no function with *int* type, then the call to ping results in calling Alice's fallback function.

```
1 contract Alice {function ping(uint) returns (uint)}
2 contract Bob {function pong(Alice c){c.ping(42);}}
```

Listing 2.12: Call to the unknown [4]

DoS by external contract — LV#4. It is possible for external calls to fail and throw exceptions or revert the transaction. Inefficient management of these calls in the contract logic can lead to critical code weaknesses, such as a Denial of Service (DoS) or loss of funds.

Timing and Optimization Weaknesses

Performance and timing code weaknesses in smart contracts usually affect the gas amount in the contracts. In the following, we define 2 code weaknesses belonging to this category.

Unbounded loops — TO#1. In Solidity, iterating through a potentially unbounded array of items can be costly, as exemplified in *getNotes()* in Listing 2.13. Since the array *notes* is provided as an input, the smart contract has no control over the maximum length, allowing a malicious actor to send in large arrays.

Creating subcontracts cost — TO#2. Contract deployments are very expensive operations. For instance, deploying a contract for every patient in Listing 2.13 is very costly. A malicious developer can use this weakness to cost the owner of the contract more Ether.

```
1 contract MedicalRecord {
2 struct Doctor {
3     bytes32 name;
```

[24]https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/

```
4       uint id;}
5   struct Note {
6       bytes32 title;
7       bytes32 note;}
8   function getNotes()
9       view
10      public
11      isCurrentDoctor
12      returns (bytes32[], bytes32[])
13  {
14      bytes32[] memory titles = new bytes32[](notes.length);
15      bytes32[] memory noteTexts = new bytes32[](notes.length);
16      for (uint i = 0; i < notes.length; i++) {
17          Note storage snote = notes[i];
18          titles[i] = snote.title;
19          noteTexts[i] = snote.note;
20      }
21      return (titles, noteTexts);}
```

Listing 2.13: Unbounded loops and creating subcontracts

Costly state variable data type — TO#3. Because of the padding rules, the byte[] data type consumes more gas than a byte array. In addition, declaring variables without constant consumes more gas than one declared with a constant. This weakness is also reported in [11].

Costly function type — TO#4. A function declared public rather than external and not utilized within the contract consumes more gas on deployment than it should.This weakness is also reported in [11].

Compatibility Weaknesses

This category is related to code weaknesses that prevent Ethereum from running normally on the developer machine. We find that the main root causes of this category are: (1) developers are not using the latest binaries/releases and (2) the hardware that is in use does not meet the minimum requirements. As this category played only a minor role in the analyzed posts, we did not label the posts in detail, but decided to keep this for future work.

Deployment and Configurations

This category of weaknesses is caused by wrong configurations and deployment.

Improper configuration — DL#1. Wrong or improper configuration of the smart contract application tool-chain can result in weaknesses or errors,

which applies even if the contract itself is free of code weaknesses.

Violating contract size limit — DL#2. In Ethereum, limits are imposed by the gas consumed for the transaction. While there is no exact size limit there is a block gas limit and the amount of gas provided has to be within that limit. When deploying a contract, there are three factors to consider: (1) an intrinsic gas cost, (2) the cost of the constructor execution, and (3) the cost of storing the bytecode. The intrinsic gas cost is static, but the other two are not. The more gas consumed in the constructor, the less is available for storage. Normally, the vast majority of gas consumed is based on the size of the contract. If the contract size is large, the consumed gas can get close to the block gas size limits, preventing the deployment of the contract.

Authorization and Authentication

The following weaknesses directly affect the security of a smart contract and could enable attacks/exploits.

Lack of access control management — SV#1. Access control is an essential element to the security of a smart contract. Based on the privileges of each client/contractor party, there have to be strict rules implemented in the contract that enforce access control.

For example, the contract in Listing 2.14 is trying to provide functionality to whitelist addresses. The original function in line 10 does not have any access restrictions, meaning any caller can whitelist addresses. The modified version in line 15 only allows the contract owner to do so.

```
1  contract WHITELIST {
2      address owner; //set during the first call
3      modifier isOwner() {
4          require(msg.sender == owner);
5          _;
6      }
7      // insecure
8      function enableWhitelist(address address) {
9          //Whitelist an address
10     }
11     // secure
12     function enableWhitelist(address address) external isOwner {
13         //Whitelist an address
14     }
15 }
```

Listing 2.14: Lack of access control management

Authorization via tx.origin — SV#2. tx.origin is a global variable in Solidity which returns the address of the account that sent the transaction. Rather than returning the immediate caller, tx.origin returns the address of the original sender (i.e., the first sender who initiated the transaction chain). It can make the contract vulnerable, if an authorized account calls into a malicious contract. Therefore, a call could be made to the vulnerable contract that passes the authorization check as tx.origin returns the original sender of the transaction, which in this case is the authorized account.

Signature based code weaknesses — SV#3. These code weaknesses are introduced as a result of insufficient signature information or weaknesses in signature generation and verification. Those include but not limited to:

- Lack of proper signature verification: One example can be relying on *msg.sender* for authentication and assuming that if a message is signed by the sender address, then it has also been generated by the sender address. Particularly in scenarios where proxies can be used to relay transactions, this can lead to code weaknesses. For more information, we refer to SWC-122[25]

- Missing Protection against Signature Replay Attacks: To protect against Signature Replay Attacks, a secure implementation should keep track of all processed message hashes and only allows new message hashes to be processed. Without such control, it would be possible for a malicious actor to attack a contract and receive message hashes that were sent by another user multiple times. For more information, we refer to SWC-121[26]

- Signature Malleability: The implementation of a cryptographic signature system in Ethereum contracts often assumes that the signature is unique, but signatures can be altered without the possession of the private key and still be valid. Valid signatures could be created by a malicious user to replay previously signed messages. For more information, we refer to SWC-117[27]

Dependency and Upgradability Weaknesses

Dependencies of and upgrades to smart contracts can lead to a number of issues in smart contracts. We describe one of these weaknesses below.

[25] `https://swcregistry.io/docs/SWC-122`
[26] `https://swcregistry.io/docs/SWC-121`
[27] `https://swcregistry.io/docs/SWC-117`

Insecure contract upgrading — UV#1. There are two ways to upgrade a contract. First, to use a registry contract that keeps track of the updated contracts, and second, to split the contract into a logic contract and a proxy contract so that the logic contract is upgradable while the proxy contract is the same. Both approaches allow untrusted developers to introduce dependency code weaknesses in the updated contract's logic, allowing attackers to modify the logic of the upgraded contract using its dependencies, e.g. another contract. This code weakness was also reported in [4] and [13].

Interface Weaknesses

This category describes weaknesses resulting in incorrect display of results. However, in these cases the smart contracts actually worked and produced the expected outcomes. The weaknesses were instead found in other applications that displayed the results. As these applications are not part of the smart contract, and there was no apparent code weakness in the contracts, we did not further investigate these kind of weaknesses.

2.5.2 How Do Frequency Distributions Compare Across Data Sources? (RQ2)

To answer RQ2, we analyzed the frequency distributions of the defined eleven code weakness categories across the four data sources (i.e., Stack Overflow GitHub, CVE, and SWC). The frequency distributions of the defined categories are shown in Figures 2.6 and 2.7 for the four data sources. Note that Figure 2.7 (a) uses a different scale than the other three figures.

> The resulting frequency distribution shows that the Language specific coding category and the Structural data flow category are the most common code weakness categories in Ethereum smart contracts

The language specific coding category is dominant on both Stack Overflow and GitHub. In contrast, structural data flow weaknesses are most frequent on CVE and SWC. Additionally, almost 80% of the reported structural data flow weaknesses in the NVD database (CVEs) are integer overflow/ underflow weaknesses. Interestingly, issues on Stack Overflow and GitHub appear to have similar frequency distributions. However, there is a statistically significant association between the frequency distributions and the data sources according to Fisher's exact test with an alpha value of 0.05 ($p=0.011$).

We further analyzed the variations of code weakness frequencies among the studied data sources. According to our analysis, by far the most frequent smart contract weaknesses in the CVE records fall under the Structural Data Flow (SV-DF) category and specifically under the arithmetic overflow/underflow category. This type of code weakness is common in software engineering. According to a report published on the CVE website[28], this code weakness has been among the top 10 weaknesses reported to CVE records for many years. Interestingly, SV-DF weaknesses are uncommon in Stack Overflow and on SWC.

2.5.3 What Impact Do the Different Categories of Smart Contract Code Weaknesses Have? (RQ3)

In this section, we present the main impacts of vulnerabilities in smart contracts, thus answering RQ3. We unify all the proposed impact categories in the literature (i.e., [11, 54]), and present a thorough classification scheme of code weakness impacts on Ethereum smart contracts. As discussed in Section 5.3, we followed a similar approach as in answering RQ1. The definitions of the final impact categories are depicted in Table 2.6, where the upper part of the table lists categories related to the impact on the software product (I-D1), and the lower part lists categories related to the impact on business factors and the contract owners (I-D2). Furthermore, Table 2.7 shows how the impact categories in literature related to our classification. Note that IP5 (in the second-last row) does not relate to any of our categories, as the category describes smart contracts that function as intended, something that we did not include in our classification.

> Our mapping shows that the impact of smart contract code weaknesses on certain aspects has not been examined in detail. For instance, the impact on the development process and the productivity of a software development team. Additional research in this area can quantify the extent to which the weaknesses impact smart contract development, the developing team, and the development of decentralized applications based on smart contracts.

Figure 2.8 presents the frequency distribution of code weakness impacts in smart contracts. In the process of analysing the impacts of various weaknesses,

[28] `https://cve.mitre.org/docs/vuln-trends/index.html`

Table 2.6: Classification scheme of impacts

Impact	Description	Short
Unexpected behavior	Contract behaves abnormally, e.g., generating incorrect output.	UB
Unwanted functionality	Contract executes wrong functionality because of wrong logic.	UF
Long response time	Long runtime of a smart contract to any input without providing the desired output.	LRT
Data Corruption	Data becomes unreadable, unusable or inaccessible, and unexpected output is generated.	DC
Memory disclosure	Problems in the memory storage of the smart contract.	MF
Poor performance	Non-optimal resource usage in terms of gas and time.	PPC
Unexpected stop	Unexpected exit and execution stop at the point of triggering the code weakness in the code.	USP
Information disclosure	When a code weakness is exploited, sensitive information is exposed to an actor not explicitly authorized to see it.	ID
Lost Ether or assets	An exploited code weakness can lead to unauthorized actors taking over the Ether of the contract and losing it.	LEA
Locked Ether	In the situation of triggering code weaknesses in a contract, one can lose access to the contract and lock the Ether in it without having access to it again.	LE
Lost control over the contract	By exploiting code weaknesses, an unauthorized actor can take over the contract.	LC

Table 2.7: Mapping literature-based impact classifications to I-D. $\subset^*$ indicates the corresponding category in our own classification is a subset of the corresponding category in the literature marked by *. $^*\subset$ means the category in the literature is a subset of our proposed category.

Category	UB	UF	LRT	DC	MF	PPC	USP	ID	LEA	LE	LC
Functionality* [54]		=									
Performance* [54]						=					
Security [54]	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$
Serviceability *[11]	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$	$\subset^*$				
IP1 unwanted behaviors* [11]	=										
IP3 non-exploitable UB* [11]	$^*\subset$										
IP4 Errors outside program call [11]		$^*\subset$									
IP5 No errors* [11]											
IP2 UB without losses* [11]	$^*\subset$									$^*\subset$	

certain cases were encountered where determining the precise impact was challenging. As a result, these cases were designated as 'NF' (i.e., no impacts) due to the lack of clear and discernible impacts. The analysis of the impacts within the language specific coding category, as shown in Figure 2.8(a), reveals that 'unexpected stop' is the most prevalent impact category. Similarly, in the data initialization and validation category as shown in Figure 2.8(b), 'data corruption' emerges as the most prevalent impact category, followed closely by 'unexpected stop' as the second most prevalent impact. Moreover, the analysis of the impacts within the structural sequence category reveals a comparable impact distribution, specifically in the two impact categories of 'unexpected stop' and 'data corruption'. These three code weakness categories demonstrate similar prevalence, emphasizing the importance of promptly addressing the impacts associated with these weaknesses and implementing effective remediation measures. In Figure 2.8 (d) and (e), the analysis of the structural data flow and logic categories reveals that 'unwanted functionality' is the most prevalent impact within these respective categories. This finding emphasizes the importance of addressing and mitigating the risks associated with unwanted functionality to ensure the robustness of smart contracts. The Timing and Optimization category encompasses two significant impacts: poor performance and long response time as shown in Figure 2.8 (f). These impacts are primarily attributed to the extensive utilization of resources, such as gas or time. The deployment & configurations category also demonstrates the prevalent impact of long response time as shown in Figure 2.8 (h). Additionally, the compatibility and dependency & upgradability categories highlights the dominant impact category of unexpected behavior. Furthermore, within the authentication & authorization category, data corruption emerges as the most prevalent impact category, followed closely by unwanted functionality as the second most frequent category. These findings emphasize the importance of robust authentication and authorization mechanisms to mitigate the legal and technical risks associated with these impacts. Finally, we did not analyse the impacts of the Interface category, given that the identified weaknesses were attributed to external applications and not directly related to the smart contracts themselves as mentioned before.

2.6 Discussion

In the following, we discuss our findings in terms of implications and relation to existing work.

We find a substantial number of code weaknesses being discussed in social

coding platforms and existing vulnerability repositories. This clearly shows that this is an important topic to study and analyze. Because of the unique characteristics of smart contracts, e.g., immutability and gas consumption, it is important to make sure that code weaknesses with severe impacts are fixed or even detected before deploying the contract to the blockchain.

As demonstrated in our findings, existing classifications either focus on a single dimension of smart contract code weaknesses, such as the error source, or mix multiple dimensions in a single classification. Our mapping unifies these different dimensions to some extent and shows how different classification schemes relate.

In addition to mixing dimensions, the majority of existing classification schemes for smart contract code weaknesses include broad categories, such as security and availability, to which many code weaknesses can be assigned. As such, these categories do not support reasoning about the included code weaknesses, which is an important quality criterion for classifications [39]. Furthermore, broad categories might prevent orthogonality of the categories, i.e., that a single code weakness fits into a single category only. In the example mentioned above, i.e., security and availability, many code weaknesses can lead to negative effects on both, and thus could be labeled both.

The frequency distributions discussed in Section 2.5.2 show notable differences between the frequencies of found code weaknesses across different data sources. This highlights that focusing on a single source biases the resulting study. It further demonstrates that established databases such as CVE and SWC do not reflect well the topics discussed in public coding platforms such as Stack Overflow or GitHub. On CVE and SWE, it appears that reported weaknesses are primarily those that are well-known and were discovered as results of known attacks. For instance, the re-entrancy attack was discovered when the infamous DAO attack took place in 2016 [31]. On Stack Overflow and GitHub, we observe specifically that developers seem to have a poor understanding of pre-defined functions such as *view* and *pure*, and that it is hard for them to cope with continuous changes and updates in Solidity and its documentation.

On the one hand, this finding suggests that tools for verification and analysis of smart contracts is of high importance, especially focusing on the prevalent code weakness categories in our classification. On the other hand, the observed frequencies might only be a symptom of the technology maturity. Hence, these issues might become less prevalent once Solidity matures and updates become less frequent. Finally, it is important to note that SWC and CVE both are aggregating weaknesses based on reports, i.e., are derived data sources. This makes direct comparison of frequencies to those on Stack Overflow and GitHub

challenging, and might explain the differences at least in part.

Moreover, we studied the impact of smart contract code weaknesses, as these impacts can uncover significant risks in smart contracts and blockchain-based applications. Understanding the proposed dimensions and the impacts of smart contract code weaknesses per dimension is also crucial for developing effective strategies to prioritize, prevent, detect, and mitigate such code weaknesses in smart contracts. We defined 11 impacts that are divided to two main dimensions. When examining the first impact dimension (i.e., I-D1), several key impacts stand out, including unexpected stoppage, memory disclosure, data corruption, long response times, unwanted functionality, and unexpected behavior. While some of these impacts are common across different software systems, there are notable differences in the context of smart contracts. Unexpected stoppage, for instance, can be disruptive in any software application, causing crashes or halting normal operations. Similarly, data disclosure and memory corruption can compromise the integrity of sensitive information. These impacts are shared by both smart contracts and other software.

However, there are distinct aspects to consider in the context of smart contracts. Smart contracts operate on decentralized platforms such as blockchains, which introduce unique challenges. The long response times of a smart contract can disrupt the execution of transactions within a decentralized network and hinder the functioning of decentralized applications (DApps) relying on them. This distributed nature and the reliance on consensus mechanisms in smart contracts differentiate their long response times from that of other software systems. Moreover, it can affect user experience and system performance in both smart contracts and other software applications. Nevertheless, in the case of smart contracts, it may be more pronounced since they often involve financial transactions or contractual legal agreements. Delays in processing transactions due to long response times can impact the timeliness of these transactions, potentially resulting in financial losses or legal disputes.

In addition, code weaknesses in smart contracts can have significant business-related impacts and directly affect contract owners (i.e., I-D2). These impacts include information disclosure, lost control over the contract, lost Ether or assets, and locked Ether. For example, the loss of Ether or assets is a critical business impact resulting from code weaknesses in smart contracts, particularly when financial transactions or digital asset management are involved. Recovering lost assets can be challenging due to the immutability of smart contracts, potentially requiring costly dispute resolution mechanisms or legal interventions. Another impact is locked Ether, which occurs when Ether becomes inaccessible or nontransferable due to code weaknesses in the contract. This situation can

arise when a smart contract is designed to hold or manage funds, but experiences a bug that prevents the contract owner from accessing their assets. Locked Ether can have severe financial implications, hindering the contract owner's ability to manage or utilize their funds as intended. Lastly, the loss of control over the contract is another consequential impact. When a code weakness compromises the logic or execution of a smart contract, the contract owner may lose control over its operations. This loss of control can have cascading effects, affecting contractual agreements, business relationships, and even financial viability. Regaining control and rectifying the code weakness may require technical expertise, collaboration with stakeholders, and potentially renegotiating affected agreements.

Based on our findings, we can provide a number of recommendations to researchers and practitioners to improve the development of smart contracts.

First, available smart contract static and dynamic detection tools must urgently target the defined categories of code weaknesses in this paper, specifically categories that are under-represented in SWC and CVE. Our breakdown of the frequencies at which the different categories occur can help prioritize this development to target the most important categories first. Second, software engineers and Solidity language designers should work together to define patch templates for known categories of code weaknesses in smart contracts. Patch templates can be applied based on the category of code weaknesses that are discovered. These templates can be implemented using each of the defined categories' root causes in this paper. Third, there is a need to define coding best practices for smart contracts and make them available to developers to avoid some language specific coding weaknesses. Software engineers are urged to follow the coding standards and recommendations by Solidity language designers, which can improve contract quality and reduce unexpected gas loss. For instance, some implementation changes on Solidity token standards (i.e., ERC-20) can result in code weaknesses.

Based on the defined categories, software engineers can implement methods to detect weaknesses in smart contracts. For example, to detect a logic code weakness category in a smart contract, software engineers can implement a sandbox to study the behavior of the contract before deployment. Also, to detect language specific coding code weaknesses, implementing a gas calculator and adding it to static analyzers can help owners estimate the amount of gas the contract will cost them.

Based on the analyzed data from the four data sources, we believe smart contract coding recommendations should include, e.g., avoiding multidimensional arrays, or arrays in general; carefully checking gas consumption amounts;

carefully checking the gas cost for each predefined function; avoid deploying big sized contracts; using specific safe libraries such as SafeMath[29] to avoid common pitfalls such as underflows; and avoiding substantial subcontract creation. We also see a need for action from the Solidity team, in particular when it comes to clearly defining gas requirements for language specific coding category including language predefined functions, declaration types, creating subcontracts, upgrading contracts, and other language artifacts and operations. Such documentation could substantially contribute to a reduction in smart contract weaknesses in Solidity/Ethereum.

Considering the impacts of smart contract code weaknesses discussed earlier, there is a clear need for the development of automated solutions that can prioritize the resolution of code weaknesses based on their potential impacts. These solutions can assist developers in addressing significant vulnerabilities, mitigating the risk of financial losses. By focusing on resolving critical weaknesses first, developers can ensure that their smart contracts are more robust and resilient, thereby protecting the interests of contract owners and users. In addition, we encourage researchers to explore the implementation of monitoring and updating mechanisms for deployed smart contracts to proactively address emerging threats and vulnerabilities, promptly responding to reported impacts. Furthermore, it is crucial for developers and practitioners to prioritize secure coding practices in smart contract development. By promoting awareness of the potential impacts of code weaknesses and providing education on secure coding techniques, developers can enhance the overall security posture of smart contracts. This includes implementing best practices such as input validation, access control, and secure data handling, which can significantly reduce the likelihood of code weaknesses and their associated impacts.

Moreover, it is important to implement maintenance measures to ensure the continued effectiveness of the proposed taxonomies. Several key measurements can be considered in this regard. First, regular updates are crucial to keep the taxonomy up to date with emerging code weaknesses in smart contracts and their impacts. For instance, AutoMESC, an automatic framework for mining and classifying Ethereum smart contract code weaknesses and their fixes [47], continuously collects newly published code weaknesses from open smart contract projects on GitHub and reported code weaknesses from CVE. This continuous data collection ensures that the code weaknesses taxonomy remains relevant and encompasses the latest smart contract code weaknesses. Additionally, periodic

[29] `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/math/SafeMath.sol`

reviews and revisions by experts in the field further contribute to keeping the taxonomies up-to-date and accurate in capturing the ever-changing landscape of smart contract code weaknesses. Second, actively engaging smart contract developers, auditors, and researchers in the taxonomy's maintenance process allows for tapping into their diverse insights and experiences, leading to a more robust classification system. Finally, mapping historical incidents to the taxonomy's code weaknesses helps to continually assess its accuracy and coverage. We believe that implementing these measures will contribute to the ongoing relevance and usefulness of the taxonomy in addressing smart contract code weaknesses and their impacts.

Finally, as the proposed taxonomies evolve, several challenges may arise when scholars and experts attempt to enrich the taxonomy in real-world scenarios. For instance, enriching the taxonomy based on real-world smart contracts extracted from Ethereum, Stack Overflow posts, OSS projects on GitHub, and other sources presents unique challenges. One such challenge is that information about vulnerabilities or their impact in each source (i.e., GitHub, Stack Overflow) can vary significantly, making it difficult to enrich the vulnerability and impact taxonomies with the required information per vulnerability. Another significant challenge we expect is that the same vulnerability might have different names or labels within different communities, such as GitHub or Stack Overflow. This can make it hard to categorize vulnerabilities consistently. One strategy to overcome this challenge is distinguishing the root cause, which is imperative to ensure the effectiveness and accuracy of the taxonomy. Additionally, addressing the impact of vulnerabilities in smart contracts poses a significant challenge. Comprehending the impacts of each vulnerability is complicated due to the complexities of smart contract behavior on the blockchain and the potential for cascading effects. Several strategies can be employed to tackle this challenge, such as considering only the impact listed in the vulnerability description, replicating the vulnerability to trace its effects, and seeking expert opinions for accurate impact classification. Lastly, the introduction of a new category into the taxonomy presents a challenge. When a new category is added, it becomes essential to review all existing vulnerabilities and determine if any should be placed within this new category. This review process ensures that vulnerabilities are accurately categorized.

2.7 Evaluation

In this section, we evaluate our proposed classification scheme using recommendations from Usman et al. [49]. At this point, we evaluate by line of argumentation. Additionally, we are currently performing an empirical evaluation with smart contract experts, which is beyond the scope of this paper.

Initially, we discuss the utility of the classification scheme, i.e., to what extent it is fit for the intended purpose. After that, we discuss benchmarking, i.e., compare our classification scheme with existing ones in the same area. Then, we argue that our classification scheme is orthogonal in its different facets, i.e., that classes of the classification do not overlap.

Finally, we outline the methodology that we intend to employ for our empirical evaluation.

2.7.1 Utility

The main purpose of the proposed classification scheme is to enable a detailed analysis of smart contract weaknesses based on their root causes and impact. We constructed this classification based on weakness databases (SWC and CVE) and real-life smart contracts or excerpts thereof. As such, utility is given by construction, since we show that actual smart contract weaknesses could be classified in the proposed way. By using a substantial sample of weaknesses from GitHub and Stack Overflow, we are further confident that the classification is general enough that it applies to smart contract weaknesses written in Solidity, beyond the sample used for creating it.

2.7.2 Benchmarking

Our classification is developed by using data from real-life contracts and vulnerabilities databases. Furthermore, we map our categories to existing classifications. While it cannot be expected that our classification is better in every aspect, we argue that it does not suffer from shortcomings observed in the existing classifications, in particular the use of overly broad classes. In addition, our classification is more detailed than existing ones, thus allowing for more detailed analyses of smart contract weaknesses.

2.7.3 Orthogonality

In our classification, categories are orthogonal, as they have definitions and main root causes that do not overlap. For example, logic code weaknesses can be inspected by following the target logic for which the contract was constructed, whereas language-specific coding categories should be inspected by following the rules and recommendations of the target language; i.e., they cannot be inspected in the same way as logic code weaknesses. Similarly, each of the other categories has its own specific characteristics and elements. For instance, there are specific methods to check interface code weaknesses and these are different from the ones that check authorization and authentication in the contract. Accordingly, we conclude that the classification is orthogonal by design.

For our empirical evaluation, we will systematically assess five main aspects of our proposed smart contract vulnerability and impact taxonomies: taxonomy importance, relevance, comprehensiveness, effectiveness, and practicality. Our evaluation methodology comprises several key phases. Initially, we will collect data by conducting focus groups and interviewing experts from leading companies in smart contracts and blockchain. These focus groups will serve as platforms for in-depth discussions, where experts can share their perspectives on the aforementioned aspects of our proposed taxonomies. The multidisciplinary nature of these focus groups will ensure a holistic evaluation, considering viewpoints from academia and industry. During these sessions, experts will be presented with the taxonomies, dimensions, and classified vulnerability types, encouraging them to critically analyze potential strengths and limitations. Following data collection, we will conduct a comprehensive analysis, which involves categorizing and structuring the data. Subsequently, we will draw meaningful conclusions from the data and engage in a thorough discussion of our findings. This process entails identifying patterns and correlations within the data, enabling us to assess the effectiveness of our taxonomies in categorizing smart contract vulnerabilities and their impacts.

2.8 Threats to Validity

In the following, we discuss threats to the validity of our findings according to internal and external validity, as well as reliability.

2.8.1 Internal Validity

The open card sorting was used as the main method to categorize and label the posts in this work. This method is subjective and open to bias, hence two experts verified the manual labeling twice independently in order to mitigate this threat. In case of any disagreement, the two experts would discuss it until reaching to a consensus. However, we acknowledge that qualitative inquiry cannot be entirely objective.

The labeling process was time intensive. We did not account for fatigue factors, but relied on the experts doing the work in shorter iterations. Hence, there is a potential threat to validity that fatigue might have affected the quality of the labels.

An additional threat to internal validity lies in the keyword search employed during data collection. For instance, to collect Q&A posts from Stack-Overflow, we used the tags "Smart contract", "Ethereum", and "Solidity", while for GitHub we used "smart contract", "Solidity", and "Ethereum". Also, a part of the collected data with insufficient information or with information that is not related to smart contract code weaknesses written in Solidity were excluded from the analysis. This search and filtering process might have led to exclude relevant data that could have led to further insights. To mitigate this threat, we collected the data using the keywords that resulted in the maximum number of records. We also made the data publicly available so researchers and developers can verify them.

As trustworthiness of the collected posts can be a source of noise in the data, we removed negative voted questions from the dataset. This might have resulted in a systematic under-representation of certain types of code weaknesses. Given the maturity of Solidity, we deemed this step warranted to allow for data of sufficiently high quality.

2.8.2 External Validity

A threat to external validity is the continuous updates of Ethereum using the hard fork. More improvements are being added to Ethereum with each hard fork such as the EIPs [30] that ensures the energy efficiency of the proof-of-work. Furthermore, many new features are added to Solidity newer versions. Therefore, new contract code weaknesses may be introduced and some others may be resolved. This means that our classification might not generalize to newer (or older) versions of Solidity.

[30] https://github.com/ethereum/EIP

Focusing on Solidity and the Ethereum blockchain only limits the external validity of the results, as there are other blockchains and other smart contract languages that could have yielded further or different code weaknesses. We focus on Ethereum since it is the second-largest blockchain, and the largest blockchain that supports smart contracts (written in Solidity). However, additional studies into other technologies could be a valuable addition to our findings, and our results might not generalize to other blockchains or smart contract languages.

Finally, some defined weaknesses can be considered as controversial. As an example, greedy contracts are those that only receive Ether and do not send it. While this can be considered a weakness in many cases, there might be cases where the contract owner intended it, e.g., for an online game that only receives Ether as in-game purchase. Similarly, hard-coded address is a well-known code weakness. To minimize this threat, we clarified in the definition of each weakness when the weakness applies and indeed results in unexpected behavior. A further issue might arise if weaknesses are coupled with other weaknesses, resulting in unpredictable behavior. This remains a future direction and an opportunity to study the behavior of these weaknesses in combination with others.

2.8.3 Reliability

To ensure reliability of the results, we described in detail the data collection and analysis process. We systematically calculated inter-rater reliability coefficients in iterative rounds, to ensure sufficiently clear categories and agreement among two expert raters. Finally, we published the full dataset [29]. Overall, these steps should ensure reliability and enable replication of our study.

2.9 Conclusion

Due to the immaturity of blockchain technology, code weaknesses in smart contracts can have severe consequences and result in substantial financial losses. Hence, it is essential to understand code weaknesses of smart contracts. However, there exist several shortcomings in existing classifications, i.e., they focus on single dimensions, mix dimensions, propose too broad categories, or rely on single data source omitting important sources for code weaknesses. To address this gap, we extracted smart contract code weaknesses written in Solidity from a number of important data sources, classified them in terms of error source and impact, and related them to existing classifications in literature. Our findings show that language-specific coding and structural data flow are the dominant

categories of code weaknesses in Solidity smart contracts. We also find that many code weaknesses are similar to known issues in general purpose programming languages, such as integer overflow and erroneous memory management. However, the immaturity and rapid evolution of the technology and the Solidity language, and the added concept of gas furthermore adds code weaknesses, and increases the risk of attacks and financial losses. Interestingly, we find that the frequency at which the different categories occur differs widely across data sources, indicating that they should not be viewed in isolation. On a meta level, these frequency distributions can present a valuable source of information to reason about biases in studies that only use one or a few of these sources.

Our classification scheme is a further step to standardize and unify code weakness analysis in smart contracts. This can support researchers in building tools and methods to avoid, detect, and fix smart contract code weaknesses in the future.

Specifically, we see a number of directions for future research. First, we are currently in the process of performing an empirical evaluation of the taxonomy with experts in smart contracts and in security, in particular of the utility of the taxonomy for practical use. Second, future studies should investigate whether our classification scheme can be generalized by investigating other smart contracts in other blockchain networks, e.g., Hyperledger, Stellar, and Openchain. Potentially, our classification needs to be modified to fit into other networks or languages. Similarly, the classification might have to be adapted as Solidity evolves and as new languages are developed for Ethereum smart contracts. Third, more dimensions and characteristics of the studied code weaknesses can be explored in the future. For instance, patterns among the extracted code weaknesses could be abstracted, as well as the evolution over time. For each of the categories, code metrics and detection tools can be explored. Based on our results, future work should investigate in more depth why frequency distributions differ across data sources. Additionally, a valuable contribution would be to study how the weakness categories are connected with the impact categories, i.e., which weaknesses typically have which impact(s). It should be investigated in which ways the defined code weaknesses can be exploited, as well as what the impact of these exploits will be. This can be accomplished by developing automated scripts or manually devising such exploits. Finally, Ethereum merge has happened very recently with multiple significant changes, e.g., the protocol has been updated from "proof-of-work" to "proof-of-stake". This might affect existing code weaknesses and the proposed taxonomy. Thus, a follow-up study should be conducted to assess the impact of this merge on the classification, also in order to assess the robustness of our classification for future merges.

Acknowledgment

The authors would like to thank Mohammad Alsarhan, a security expert, for participating in the card sorting and inter-rater agreement discussions. This work was supported by the Icelandic Research Fund (Rannís) grant number 207156-051.

Data Availability

The datasets generated during and/or analysed during the current study are available in the Zenodo repository, `https://doi.org/10.5281/zenodo.6388179`.

```
pragma solidity >=0.7.0 <0.9.0;
contract Ballot {
    struct Voter {
        uint weight; // weight is accumulated by delegation
        bool voted;  // if true, that person already voted
        address delegate; // person delegated to
        uint vote;   // index of the voted proposal
    }
    struct Proposal {
        bytes32 name;   // short name (up to 32 bytes)
        uint voteCount; // number of accumulated votes
    }

    address public chairperson;
    mapping(address => Voter) public voters;
    Proposal[] public proposals;
    constructor(bytes32[] memory proposalNames) {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;
        for (uint i = 0; i < proposalNames.length; i++) {

            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }
    function giveRightToVote(address voter) external {
        require(
            msg.sender == chairperson,
            "Only chairperson can give right to vote."
        );
        require(
            !voters[voter].voted,
            "The voter already voted."
        );
        require(voters[voter].weight == 0);
        voters[voter].weight = 1;
    }
    function vote(uint proposal) external {
        Voter storage sender = voters[msg.sender];
        require(sender.weight != 0, "Has no right to vote");
        require(!sender.voted, "Already voted.");
        sender.voted = true;
        sender.vote = proposal;
        proposals[proposal].voteCount += sender.weight;
    }
```

Figure 2.1: Voting contract example written in Solidity [19]

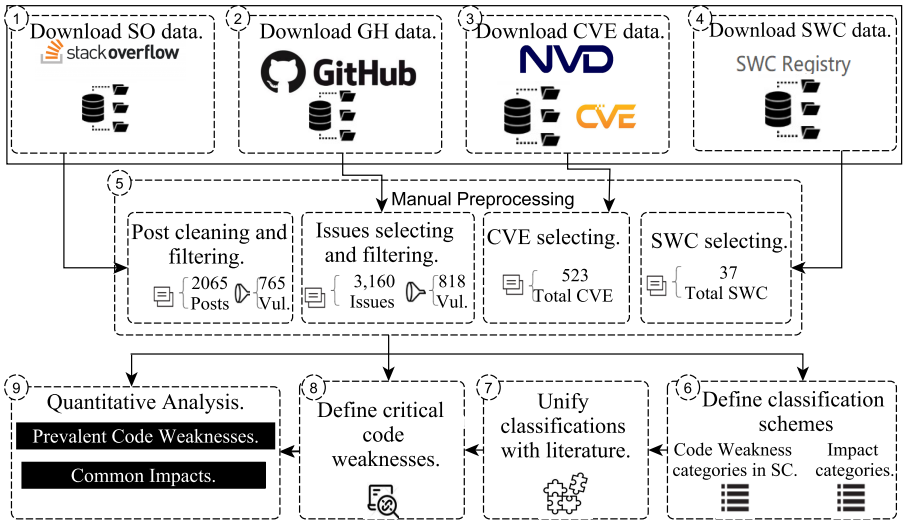


Figure 2.2: Empirical study method

README.md

Vulnerability Type

Typo in Constructor

Abstract

We found a vulnerability in smart contract of "MORPH" token. Because there is a typ
MORPH token, allows attackers to acquire contract ownership. A new owner can sub
DoS attack.

Details

'MORPH' is an Ethereum ERC20 Token contract. The total number of transactions sul
token. Moreover, the last transaction date of this contract is 3 days ago which is acti

Figure 2.3: Code weakness example from GitHub

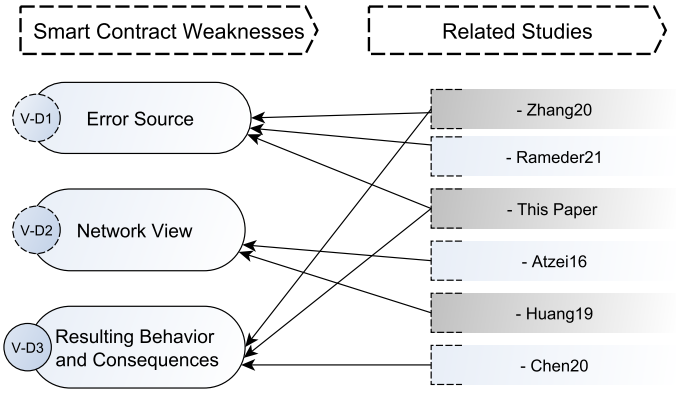


Figure 2.4: Dimensions of smart contract code weaknesses (*V-D*)

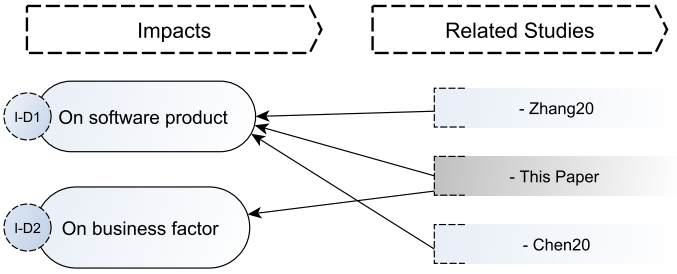
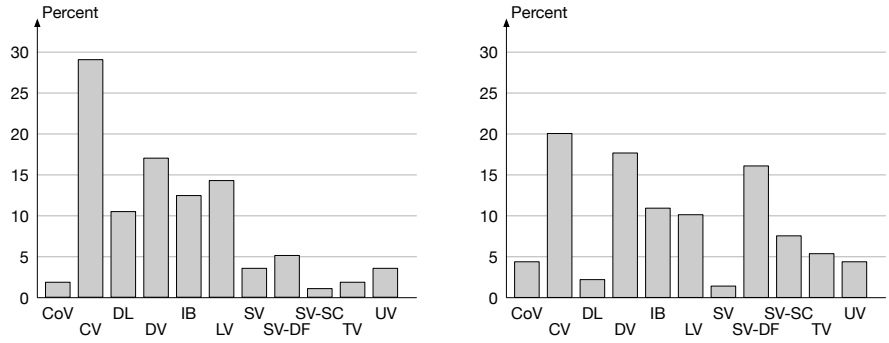
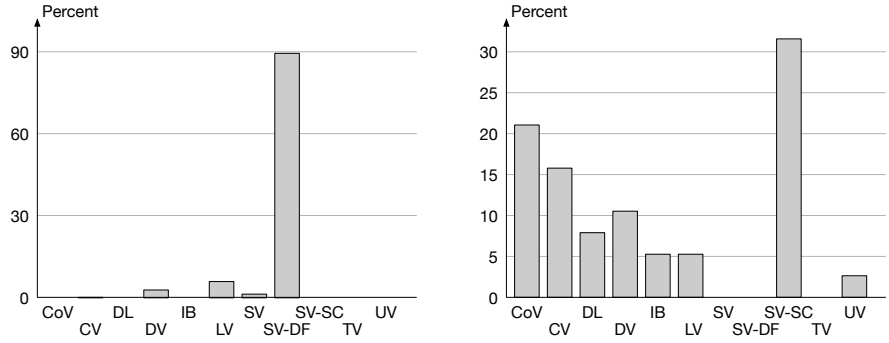


Figure 2.5: Dimensions of smart contract code weakness impacts (*I-D*)



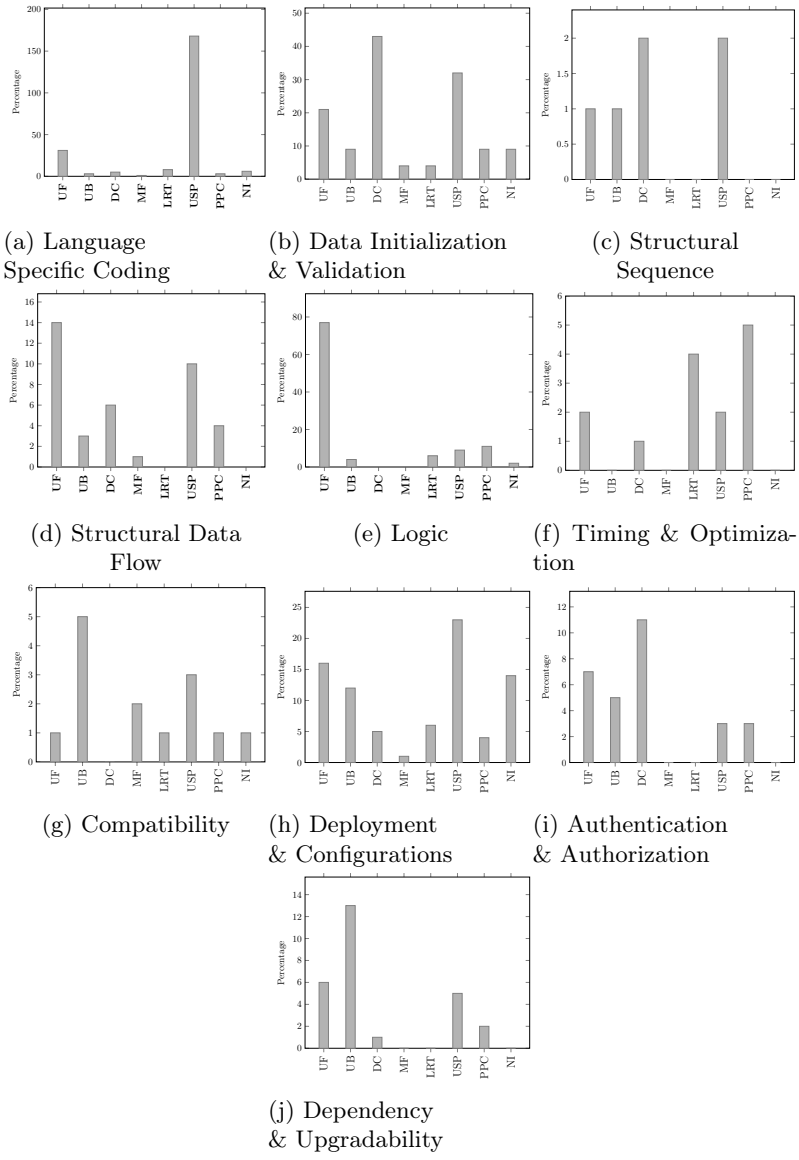
(a) Stack Overflow (b) GitHub

Figure 2.6: Code weakness frequency distribution in (a) StackOverflow and (b) GitHub



(a) CVE (b) SWC

Figure 2.7: Code weakness frequency distribution in (a) CVE and (b) SWC



(a) Language Specific Coding

(b) Data Initialization & Validation

(c) Structural Sequence

(d) Structural Data Flow

(e) Logic

(f) Timing & Optimization

(g) Compatibility

(h) Deployment & Configurations

(i) Authentication & Authorization

(j) Dependency & Upgradability

Figure 2.8: Frequency distribution of code weakness impacts per code weakness category in smart contracts

Bibliography

[1] Ahasanuzzaman, M., Asaduzzaman, M., Roy, C.K., Schneider, K.A.: Mining duplicate questions of stack overflow. In: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), pp. 402–412. IEEE (2016) 66

[2] Alharby, M., Aldweesh, A., van Moorsel, A.: Blockchain-based smart contracts: A systematic mapping study of academic research (2018). In: 2018 International Conference on Cloud Computing, Big Data and Blockchain (ICCBB), pp. 1–6. IEEE (2018) 62

[3] Alharby, M., Van Moorsel, A.: Blockchain-based smart contracts: A systematic mapping study. arXiv preprint arXiv:1710.06372 (2017) 4, 54, 62, 65, 75

[4] Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts. IACR Cryptol. ePrint Arch. **2016**, 1007 (2016) 3, 13, 54, 62, 65, 75, 80, 85

[5] Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). In: International conference on principles of security and trust, pp. 164–186. Springer (2017) 1, 53, 89, 126, 150, 194, 196, 215

[6] Ayman, A., Aziz, A., Alipour, A., Laszka, A.: Smart contract development in practice: Trends, issues, and discussions on stack overflow. arXiv preprint arXiv:1905.08833 (2019) 66

[7] Ayman, A., Roy, S., Alipour, A., Laszka, A.: Smart contract development from the perspective of developers: Topics and issues discussed on social

media. In: M. Bernhard, A. Bracciali, L.J. Camp, S. Matsuo, A. Maurushat, P.B. Rønne, M. Sala (eds.) Financial Cryptography and Data Security, pp. 405–422. Springer International Publishing, Cham (2020) 66

[8] Badawi, E., Jourdan, G.V.: Cryptocurrencies emerging threats and defensive mechanisms: A systematic literature review. IEEE Access (2020) 57

[9] Bajaj, K., Pattabiraman, K., Mesbah, A.: Mining questions asked by web developers. In: Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 112–121 (2014) 66

[10] Bhat, M., Vijayal, S.: A probabilistic analysis on crypto-currencies based on blockchain. In: 2017 International Conference on Next Generation Computing and Information Systems (ICNGCIS), pp. 69–74. IEEE (2017) 57

[11] Calefato, F., Lanubile, F., Marasciulo, M.C., Novielli, N.: Mining successful answers in stack overflow. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, pp. 430–433. IEEE (2015) 66

[12] Chen, C., Xing, Z.: Mining technology landscape from stack overflow. In: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 1–10 (2016) 66

[13] Chen, H., Pendleton, M., Njilla, L., Xu, S.: A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. ACM Computing Surveys (CSUR) **53**(3), 1–43 (2020) 3, 14, 62, 63, 65, 71, 89, 193

[14] Chen, J., Xia, X., Lo, D., Grundy, J., Luo, X., Chen, T.: Defining smart contract defects on ethereum. IEEE Transactions on Software Engineering (2020) 3, 13, 53, 54, 63, 66, 75, 86, 90, 92, 178

[15] Cosentino, V., Izquierdo, J.L.C., Cabot, J.: A systematic mapping study of software development with github. IEEE Access **5**, 7173–7192 (2017) 67, 158

[16] Daian, P.: Analysis of the dao exploit. Hacking, Distributed **6** (2016) 62

[17] Dingman, W., Cohen, A., Ferrara, N., Lynch, A., Jasinski, P., Black, P.E., Deng, L.: Defects and vulnerabilities in smart contracts, a classification using the nist bugs framework. International Journal of Networked and Distributed Computing **7**(3), 121–132 (2019) 53, 54, 62, 63

[18] Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P.: Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 530–541 (2020) 14, 67, 119, 121, 122, 134, 135, 158, 196, 226

[19] Ethereum: Solidity by example (2021). URL `https://docs.soliditylang.org/en/latest/solidity-by-example.html` xiii, 8, 60, 105

[20] Ethereum Improvement Proposals (EIPs): EIP-1470: Smart Contract Weakness Classification (SWC). `https://github.com/ethereum/EIPs/issues/1469`. [Online] Accessed: 01-April-2020 56, 150

[21] Hewa, T., Ylianttila, M., Liyanage, M.: Survey on blockchain based smart contracts: Applications, opportunities and challenges. Journal of Network and Computer Applications p. 102857 (2020) 53

[22] Huang, Y., Bian, Y., Li, R., Zhao, J.L., Shi, P.: Smart contract security: A software lifecycle perspective. IEEE Access **7**, 150184–150202 (2019) 13, 61, 62

[23] Idelberger, F., Governatori, G., Riveret, R., Sartor, G.: Evaluation of logic-based smart contracts for blockchain systems. In: International symposium on rules and rule markup languages for the semantic web, pp. 167–183. Springer (2016) 61

[24] Kalra, S., Goel, S., Dhawan, M., Sharma, S.: Zeus: Analyzing safety of smart contracts. In: Ndss, pp. 1–12 (2018) 61

[25] Khan, S.N., Loukil, F., Ghedira-Guegan, C., Benkhelifa, E., Bani-Hani, A.: Blockchain smart contracts: Applications, challenges, and future trends. Peer-to-peer Networking and Applications **14**(5), 2901–2925 (2021) 53

[26] Khan, Z.A., Namin, A.S.: Ethereum smart contracts: Vulnerabilities and their classifications. In: 2020 IEEE International Conference on Big Data (Big Data), pp. 1–10. IEEE (2020) 57

[27] Khan, Z.A., Namin, A.S.: A survey on vulnerabilities of ethereum smart contracts. arXiv preprint arXiv:2012.14481 (2020) 64

[28] Maiden, N.: Card sorts to acquire requirements. IEEE software **26**(3), 85–86 (2009) 69

[29] Majd Soud Grischa Liebel, M.H.: Dataset: A fly in the ointment: An empirical study on the characteristics of ethereum smart contracts code weaknesses and vulnerabilities (2021). doi:10.5281/zenodo.4441254. URL `https://doi.org/10.5281/zenodo.6388179` 55, 73, 102

[30] Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: Verisolid: Correct-by-design smart contracts for ethereum. In: International Conference on Financial Cryptography and Data Security, pp. 446–465. Springer (2019) 61

[31] Mehar, M.I., Shier, C.L., Giambattista, A., Gong, E., Fletcher, G., Sanayhie, R., Kim, H.M., Laskowski, M.: Understanding a revolutionary and flawed grand experiment in blockchain: the dao attack. Journal of Cases on Information Technology (JCIT) **21**(1), 19–32 (2019) 27, 53, 94

[32] Nawaz, A.: A comparison of card-sorting analysis methods. In: 10th Asia Pacific Conference on Computer Human Interaction (Apchi 2012). Matsuecity, Shimane, Japan, pp. 28–31 (2012) 69

[33] Norvill, R., Fiz, B., State, R., Cullen, A.: Standardising smart contracts: Automatically inferring erc standards. In: 2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), pp. 192–195. IEEE (2019) 53

[34] Perez, D., Livshits, B.: Smart contract vulnerabilities: Vulnerable does not imply exploited. arXiv preprint arXiv:1902.06710 (2019) 56, 150

[35] Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.: Verx: Safety verification of smart contracts. In: 2020 IEEE symposium on security and privacy (SP), pp. 1661–1677. IEEE (2020) 61

[36] Philosophy, G.: Solidity by example (2019). URL `https://consensys.github.io/smart-contract-best-practices/general_philosophy/` 61

[37] Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R., Lanza, M.: Mining stackoverflow to turn the ide into a self-confident programming prompter. In: Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 102–111 (2014) 66

[38] Praitheeshan, P., Pan, L., Yu, J., Liu, J., Doss, R.: Security analysis methods on ethereum smart contract vulnerabilities: a survey. arXiv preprint arXiv:1908.08605 (2019) 14, 62, 63

[39] Ralph, P.: Toward methodological guidelines for process theories and taxonomies in software engineering. IEEE Transactions on Software Engineering **45**(7), 712–735 (2018) 53, 54, 94

[40] Rameder, H.: Systematic review of ethereum smart contract security vulnerabilities, analysis methods and tools. Ph.D. thesis, Wien (2021) 53, 61, 64, 65, 75, 119

[41] Rugg, G., McGeorge, P.: The sorting techniques: a tutorial paper on card sorts, picture sorts and item sorts. Expert Systems **14**(2), 80–93 (1997) 69

[42] Sakai, R., Aerts, J.: Card sorting techniques for domain characterization in problem-driven visualization research. In: EuroVis (Short Papers), pp. 121–125 (2015) 69

[43] Samreen, N.F., Alalfi, M.H.: A survey of security vulnerabilities in ethereum smart contracts. arXiv preprint arXiv:2105.06974 (2021) 14, 63

[44] Sánchez, D.C.: Raziel: Private and verifiable smart contracts on blockchains. arXiv preprint arXiv:1807.09484 (2018) 62

[45] Seacord, R.C., Householder, A.D.: A structured approach to classifying security vulnerabilities. Tech. rep., Carnegie-mellon university pittsburgh pa software engineering inst (2005) 53

[46] Sillaber, C., Waltl, B., Treiblmaier, H., Gallersdörfer, U., Felderer, M.: Laying the foundation for smart contract development: an integrated engineering process model. Information Systems and e-Business Management **19**(3), 863–882 (2021) 53

[47] Soud, M., Qasse, I., Liebel, G., Hamdaqa, M.: Automesc: Automatic framework for mining and classifying ethereum smart contract vulnerabilities and their fixes. arXiv preprint arXiv:2212.10660 (2022) 97

[48] Spencer, D.: Card sorting: Designing usable categories. Rosenfeld Media (2009) 68, 69

[49] Usman, M., Britto, R., Börstler, J., Mendes, E.: Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method. Information and Software Technology **85**, 43–59 (2017) 99

[50] Vacca, A., Di Sorbo, A., Visaggio, C.A., Canfora, G.: A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges. Journal of Systems and Software **174**, 110891 (2021) 65, 119

[51] Vegas, S., Juristo, N., Basili, V.R.: Maturing software engineering knowledge through classifications: A case study on unit testing techniques. IEEE Transactions on Software Engineering **35**(4), 551–565 (2009) 53

[52] Viera, A.J., Garrett, J.M., et al.: Understanding interobserver agreement: the kappa statistic. Fam med **37**(5), 360–363 (2005) 70, 169

[53] Wan, Z., Xia, X., Lo, D., Chen, J., Luo, X., Yang, X.: Smart contract security: A practitioners' perspective. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 1410–1422. IEEE (2021) 61

[54] Wood, J.R., Wood, L.E.: Card sorting: current practices and beyond. Journal of Usability Studies **4**(1), 1–6 (2008) 69

[55] Yamashita, K., Nomura, Y., Zhou, E., Pi, B., Jun, S.: Potential risks of hyperledger fabric smart contracts. In: 2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE), pp. 1–10. IEEE (2019) 61

[56] Zhang, P., Xiao, F., Luo, X.: A framework and dataset for bugs in ethereum smart contracts. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 139–150. IEEE (2020) 3, 13, 14, 15, 53, 54, 62, 63, 64, 65, 75, 90, 92, 119, 121, 122, 134, 135, 178, 199, 203, 205, 206, 213, 224

[57] Zheng, Z., Xie, S., Dai, H.N., Chen, W., Chen, X., Weng, J., Imran, M.: An overview on smart contracts: Challenges, advances and platforms. Future Generation Computer Systems **105**, 475–491 (2020) 53, 119

Chapter 3

Classifying Smart Contract Vulnerabilities and Their Fixes

AutoMESC: Automatic Framework for Mining and Classifying Ethereum Smart Contract Vulnerabilities and Their Fixes

Soud, M., Qasse, I., Liebel, G., and Hamdaqa, M. 2023.

Proceedings of the 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2023).

Abstract

Due to the risks associated with vulnerabilities in smart contracts, their security has gained significant attention in recent years. However, there is a lack of open datasets on smart contract vulnerabilities and their fixes that allows for data-driven research. Towards this end, we propose an automated framework for mining and classifying Ethereum's smart contract vulnerabilities and their corresponding fixes from GitHub and from the Common Vulnerabilities and Exposures (CVE) records in the National Vulnerability Database. We implemented the proposed method in a fully automated framework, which we call AutoMESC. AutoMESC uses seven of the most well-known smart contract security tools to classify and label the collected vulnerabilities based on vulnerability types. Furthermore, it collects metadata that can be used in data-intensive smart contract security research (e.g., vulnerability detection, vulnerability classification, severity prediction, and automated repair). We used AutoMESC to construct a sample dataset and made it publicly available. Currently, the dataset contains 6.7K smart contract vulnerability-fix pairs written in Solidity. We assess the quality of the constructed dataset in terms of accuracy, provenance, and relevance, and compare it with existing datasets. AutoMESC is designed to collect data continuously and keep the corresponding dataset up-to-date with newly discovered smart contract vulnerabilities and their fixes from GitHub and CVE records.

3.1 Introduction

Smart contracts are computerized self-executing contracts that contain clauses that are enforced once certain conditions are met [12]. The concept of smart contract was first introduced in 1996 by Szabo [46]. However, due to technology limitations at that time, the first real-world implementation of a smart contract was developed by the Ethereum blockchain platform in 2014 [6].

The evolution of smart contracts has enabled blockchain technology to grow rapidly and has led to the adoption of decentralized applications in various fields such as the Internet of Things (IoT), supply chain management, and identity management [35]. While smart contracts have the potential to reshape how businesses are carried out, several challenges need to be addressed [35] that are not commonly found in traditional software development [15, 32]. For instance, immutability of smart contracts [54] can lead to financial losses, such as the infamous DAO attack that led to a theft of approximately 50 million US dollars [43].

To improve the security of smart contracts, several methods and tools have been suggested to detect and fix security vulnerabilities in smart contract source code [23]. For evaluation, these contributions rely on existing datasets [2, 7]. However, existing datasets, e.g., [27, 16, 54, 24], are limited in one of the following ways: (i) collecting and labeling the data is time and resource-intensive, e.g., [27], (ii) they are limited in terms of the amount of metadata they contain, e.g., [16], (iii) they are incompletely labeled or classified, e.g., [16], (iv) fixes of the labeled vulnerabilities are not considered, e.g, [24], and (v) they are not updated regularly, over time resulting in invalid or outdated data, e.g. [54].

To address these limitations, we propose a method to automatically mine, classify and label smart contract vulnerabilities and corresponding fixes from GitHub[1] and CVE[2] records. We implement the proposed method in a fully automated framework, AutoMESC, that curates and classifies the collected vulnerabilities based on their Common Weakness Enumeration(CWE) and vulnerability types using 7 of the most well-known smart contract security tools in the literature. We target Ethereum smart contracts written in Solidity[3] and Vyper[4], the two most common languages for Ethereum smart contracts. AutoMESC places an emphasis on extracting a large number of attributes, so that the resulting datasets can be used for use cases that rely on large number of

[1] https://github.com/
[2] https://cve.mitre.org/
[3] https://docs.soliditylang.org/en/v0.8.13/
[4] https://vyper.readthedocs.io/en/stable/

labeled data, e.g., machine learning (ML) applications.

Contributions. To summarize the most salient contributions of our research, we:

1. provide a comprehensive survey and comparison of the available datasets for smart contract vulnerabilities written in the two most popular Ethereum languages, Solidity and Vyper. To the best of our knowledge, this is the first study of existing Ethereum smart contract vulnerability datasets.

2. present a method for mining and classifying smart contract vulnerabilities and corresponding fixes from GitHub and CVE records. We implement the proposed method in an automated framework (i.e., AutoMESC).

3. present a sample dataset with statistics extracted using AutoMESC. Finally, we evaluate the quality of this dataset in terms of accuracy, relevance, and provenance.

The dataset and AutoMESC framework are publicly available at `https://github.com/majdsoud/AutoMESC-Framework`.

3.2 Related Work

This section surveys existing smart contract vulnerability datasets, related tools, and existing research gaps.

3.2.1 Smart Contracts Vulnerability Datasets

Numerous smart contract vulnerability-related datasets have been developed over the last few years. Each of the proposed datasets are unique and have different strengths and weaknesses. We identified relevant datasets using Google's dataset search engine[5] and the keywords *"smart contract vulnerabilities"*, *"smart contract fixes"*, and *"smart contracts"*[6]. We excluded datasets not related to Ethereum smart contract vulnerabilities and considered only datasets with published papers/preprints or a public GitHub repository. Finally, we excluded datasets that only contained sample contracts, as they are not relevant to the scope of this paper. Detailed information about the included and excluded

[5] `https://datasetsearch.research.google.com/`

[6] At the end of March 2022.

datasets is available publicly [7]. We included five studies and corresponding datasets:

Yashavant et al. [27] constructed a dataset called ScrawlD of real world Ethereum smart contracts labeled with vulnerabilities. It was created to support unbiased evaluation of existing tools for analyzing smart contracts. The dataset has 6.7k labeled Ethereum smart contracts extracted from Etherscan [8].

Durieux et al. [16] presented two novel datasets with the goal of evaluating the precision of smart contract analysis tools. The first dataset consists of 69 annotated vulnerable smart contracts and the second consists of 47,518 contracts extracted from Etherscan. The first dataset is curated and labeled with the location and category of the vulnerabilities, while the second dataset consists of smart contract samples. Since we consider datasets with vulnerabilities and/or fixes, we only include the first dataset.

Ren et al. [24] constructed a dataset with 46,186 contracts crawled from Etherscan, SolidiFI repository, CVE and Smart Contract Weakness Classification and Test Cases library, only some of which are labeled. The dataset is available on GitHub[9]. The labeled dataset has a total of 350 artificially constructed contracts and 214 confirmed vulnerable contracts.

Zhang et al. [54] constructed a dataset called Jiuzhou. It consists of 176 smart contracts with vulnerabilities. It provides two contracts for every type of the studied vulnerability: one with the vulnerability and one without it.

Gigahorse benchmarks[10] consists of a collection of Ethereum smart contracts in source and binary format, labeled with respective vulnerabilities. Some contracts in this dataset have been derived from Durieux et al. [16].

3.2.2 Automated Tools

This section presents an overview of the related automated tools to AutoMESC including general purpose tools for mining vulnerabilities.

Ferreira et al. [11] introduced SmartBugs, an automated framework to analyze smart contracts written in Solidity based on 19 security analysis tools. The tool does not collect or construct datasets. The framework was used to label the aforementioned dataset in [16].

CVEfixes [2] is a dataset and an automated collection tool that collects vulnerable code and corresponding fixes from open-source software repositories.

[7] https://doi.org/10.5281/zenodo.6762730

[8] etherscan.io

[9] https://github.com/renardbebe/Smart-Contract-Benchmark-Suites

[10] https://github.com/nevillegrech/gigahorse-benchmarks

CVEfixes supports data collection from multiple programming language repositories. However, the CVEfixes dataset does not contain any vulnerable smart contract code. In addition, CVEfixes relies solely on CVE to classify code vulnerabilities. This is not enough in the case of smart contracts, as there are very few CVE records for smart contracts. Furthermore, smart contracts have unique vulnerabilities such as re-entrancy, integer arithmetic errors, or extra gas consumption. These vulnerabilities can not be detected by traditional general-purpose security tools, and require specialized tools in smart contracts to label and classify them.

3.2.3 Research Gaps

Based on the related datasets and frameworks, we identified the following research gaps:

Datasets that support data-driven approaches are scarce. Most datasets were not created for supporting data-driven research in the field of smart contracts vulnerabilities or fixes. Existing datasets are constructed with the purpose of evaluating state-of-the-art tools. For instance, [27] and [24] aim to eliminate bias in assessing smart contracts security analysis tools. Also, [16] and [54] support the evaluation of smart contract security analysis tools.
Most available datasets are not labeled or contain only a few labeled vulnerable contracts. These are very general datasets, where researchers need to conduct labeling and pre-processing to utilize the data with data-driven models. Therefore, there is a need for datasets with diverse samples of smart contract vulnerabilities and their fixes for reliable training and evaluation of ML approaches [25, 8, 13].
Existing datasets are not updated regularly. Several vulnerability types become deprecated over time, and there is a need to collect vulnerabilities that developers encounter on a day-to-day basis.
In the labeled datasets, fixes are not addressed. There is a lack of databases covering smart contract vulnerability fixes or the relationships among them. This limits research that requires both vulnerable and fixed code. **There is no variation in the level of granularity of labeled datasets**, since most of them are at the contract level. According to [36], levels of granularity and the precise location of a vulnerability in the dataset are finer than the widely used granularity of programs and files. Also, [28] conclude that file-level granularity decreases precision and recall performance of analysis tools. Thus, multiple granularity levels are needed to support data-driven research on smart contracts.
There is a lack of datasets that support Vyper Ethereum smart con-

tract language research. In our review, we only found one dataset containing smart contracts written in Vyper [33]. However, at the time of writing this paper there is no publicly available labeled dataset of Vyper smart contract vulnerabilities. This is an important shortcoming, since there are many recently disclosed Vyper-related vulnerabilities with high severity in the NVD database e.g. CVE-2022-24845[11].

To address these research gaps, AutoMESC (1) classifies and labels smart contracts vulnerabilities in various ways, (2) supports a broad range of smart contract vulnerability types (i.e., 36 types), (3) includes vulnerable smart contract code and the corresponding fixes (n=6.7K) at different levels of granularity, and (4) supports regular updates.

3.3 Details of AutoMESC

In this section, we describe the proposed method to mine, classify and label smart contract vulnerabilities and fixes as a fully automated framework, AutoMESC.

[11] `https://nvd.nist.gov/vuln/detail/CVE-2022-24845`

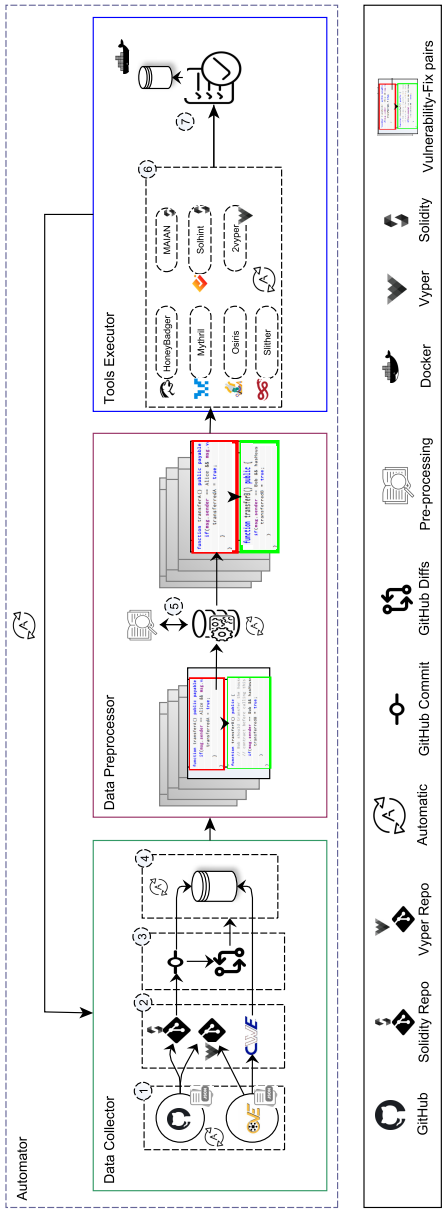


Figure 3.1: AutoMESC framework high-level architecture

Figure 3.1 illustrates the high-level architecture of AutoMESC, which is composed of three main components: Data Collector, Data Preprocessor, and Tool Executor. The Data Collector (DC) mines existing repositories to collect data from GitHub and CVE records and stores the extracted data in a relational database. The Data Preprocessor (DP) pre-processes stored data. The Tool Executor (TE) runs seven vulnerability detection tools on the data to label the vulnerabilities and finally stores the final data.

Results are stored in the Database, and the Automator component automates the process of collecting and analyzing data.

3.3.1 Data Collection Methodology

The data flow process (left-hand side of Figure 3.1) starts with automatic mining and collection of data from open-source software (OSS) projects on GitHub and NVD. Using the GitHub API, it automatically collects vulnerability-fix pairs that developers have contributed, written in Solidity or Vyper. It also automatically collects vulnerability records (CVEs) using the JSON vulnerability feed published by the NVD database, organized by the year of origin. In order to keep up to date with newly discovered and patched smart contract vulnerabilities, the automatic collection is continuously repeated from both OSS projects hosted on GitHub and the NVD database.

Mining OSS projects

The main source of smart contract vulnerabilities and corresponding fixes in AutoMESC is GitHub. AutoMESC starts by extracting all the projects hosted on GitHub with either Solidity or Vyper as their main language. AutoMESC then retrieves all related metadata for the projects and stores them in the database. AutoMESC collects all commits related to vulnerabilities and fixes using regular expressions based on selected keywords and related to *"smart contract"*, *"Solidity"*, *"Vyper"* and *"Ethereum"*, based on the common keywords associated with vulnerabilities presented by Bosu et al. [4].All commit fixes are collected and then filtered automatically, as described next in the pre-processing methodology. Vulnerable code is automatically mapped to a corresponding fix via a hash, resulting in vulnerability-fix pairs that contain both the code before and after correction. Finally, AutoMESC collects the files and methods in which the vulnerability and the corresponding fix occurred, and maps them together. This results in multiple levels of granularity, i.e., on file, method, and line level.

Mining CVE Records

AutoMESC collects all published CVEs related to Ethereum smart contracts until the last published CVE on the date of collection by retrieving the published JSON feeds from the NVD server (i.e., Step 1 in Figure 3.1). It then aggregates and processes the JSON files to remove duplicate CVEs. Afterwards, it collects and stores details about each vulnerability CVE such as CWE, published date, last modified date, and CVSS severity score (i.e., Step 2 in Figure 3.1). More details about the collected information are listed in Section 3.4.

Automation

The framework runs in the background and automatically collects newly posted vulnerabilities from OSS projects on GitHub and newly disclosed CVEs on the NVD database every two hours. The first time AutoMESC is run, it collects all data from 2016[12] onwards.

3.3.2 Data Preprocessing Methodology

AutoMESC preprocesses the collected data (center of Figure 3.1) from GitHub automatically on commit and on code level. On commit level, it excludes any commits that have no code and only considers commits that affect Solidity (.sol) and Vyper (.vy) files. AutoMESC removes comments and white spaces automatically from the files, before storing them in the database.

3.3.3 Tool Execution Methodology

AutoMESC executes (right-hand side of Figure 3.1) vulnerability detection tools on the collected data and labels the data based on the output of the tools.

Selected Tools

In order to label the collected vulnerabilities in AutoMESC, we employ available smart contract state-of-the-art analysis tools for both Solidity and Vyper (as discussed in Section 3.2). We selected 7 tools, as shown in Table 3.1, that:

- detect vulnerabilities in smart contracts and identify their types.
- take as input either a Solidity (.sol) or Vyper (.vy) source code file.

[12]The first attack on smart contracts was the DAO attack in 2016 [4].

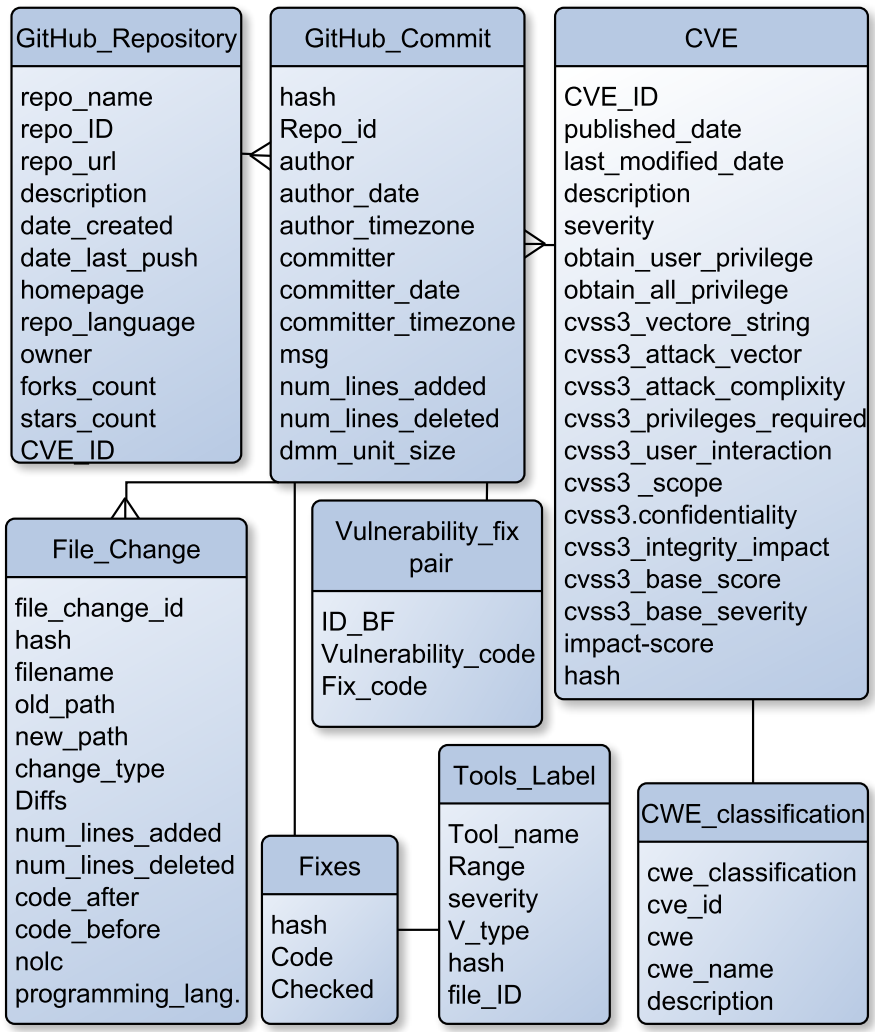


Figure 3.2: AutoMESC dataset schema

Table 3.1: Sample of supported vulnerabilities and the different labels used by each supporting tool

Vulnerability/ Tools	Osiris[49]	Slither[19]	Smart Check[48]	Solhint [34]	Honeybadger [31]	Mythril [29]	Maian [31]
Suicidal Contract	-	Suicidal	-	Avoid-suicide	-	Suicide	Suicidal contract
Integer Overflow	Arithmetic Bugs	-	Unchecked math	-	-	Integer	-
Frozen Ether	-	Locked-ether	Locked money	-	-	-	Greedy contracts
Honeypot	-	-	-	-	Honeypot	-	-

- are publicly available and based on Docker. Docker facilitates the portability of our framework and the scalability of our contract analysis.

Vulnerability Labeling

AutoMESC can detect up to 36 unique vulnerabilities based on the selected tools. However, some tools detect the same vulnerability using different names. We mapped the tools' vulnerabilities to the proposed vulnerability type classification in [52] to unify vulnerability labeling. Table 3.1 shows a sample of supported vulnerabilities and the unified label used by AutoMESC. The symbol " - " indicates that the tool does not detect the vulnerability.

Each tool selected for vulnerability detection in AutoMESC is neither sound nor complete. In some cases, a tool may generate a high number of false positives or false negatives. Therefore, we use the majority rule to label if a vulnerability exists in the data or not.

The process of labeling data as a vulnerability includes the following steps:

1. Collect the output of all selected tools for each collected data.
2. For each output, identify the detected vulnerability name and the location of the vulnerability (line number).
3. Based on the mapping in Table 3.1, unify the vulnerability name.
4. Apply the majority rule to label the collected data. AutoMESC will label a vulnerability if at least 50% of the tools that detect this vulnerability report the same vulnerability at the same position.

AutoMESC uses keyword matching to detect commits fixing vulnerabilities. In practice, this method could generate considerable noise. To handle this type of noise, AutoMESC analyses the fixed versions using the selected tools and checks whether the same vulnerability is detected or not. Utilizing the majority

rule, if the same vulnerability is found in the fixed code, AutoMESC considers this commit as noise and does not include it in the dataset as a vulnerability-fix pair.

3.4 AutoMESC Dataset

This section describes the collected data from each data source. Furthermore, we explore the constructed dataset. AutoMESC primarily collects data from the JSON vulnerability files from both NVD and projects on GitHub. Figure 3.2 depicts the overall data schema of our dataset, including the meta-data collected from NVD and GitHub.

3.4.1 Data Collected from CVE Records

Each vulnerability has a *CVE_ID*. It also has a *published_date*, *description*, *user_privilege*, *user_interaction* and *last_modified_date*. In addition, it has a severity ranking based on the Common Vulnerability Scoring System (CVSS) [18]. The NVD database provides two versions of the CVSS, CVSSv2 and CVSSv3. We store CVSSv3, as it is possible to differentiate between different types of vulnerabilities more accurately than CVSSv2. Figure 3.2 shows related data and meta-data attributes that are extracted from the CVE's JSON for each vulnerability in our dataset under the class *CVE*. Each vulnerability is linked with other classes in the dataset via a unique hash. It is directly linked with the commit class, as each CVE vulnerability has relevant source code repositories on GitHub and relevant commits that contain vulnerability source code and the corresponding fixes. In addition to labeling the vulnerabilities based on their CVSS, AutoMESC also classifies each vulnerability in the collected CVE according to CWE weakness types. Therefore, AutoMESC collects the details of each CWE type associated with each collected CVE record. The collected meta-data of each CWE includes its name, description, and URL. Finally, each vulnerability in the CVE is associated with reference links to OSS repositories with commits that have the vulnerability and corresponding fixes. AutoMESC visits these links and provisionally clones the repositories to collect the related commits, vulnerability and corresponding fixes, and stores them all in the database based on the unique hash of the commit listed in the CVE record.

3.4.2 Data Collected from GitHub Repositories

Each repository on GitHub with vulnerabilities in Solidity or Vyper has a unique ID (i.e., *repo_ID*). AutoMESC extracts the name, description, homepage, date of creations (i.e., *date_created*), *owner*, the date of the last push (i.e, *date_last_push*) and other meta-data for each relevant OSS repository. Such meta-data can be used to focus on specific characteristics of relevant OSS repositories, such as the total number of vulnerabilities and corresponding fixes since the creation date of the repository. In order to avoid confusing Solidity repositories and Vyper repositories, AutoMESC also extracts the repository language. Finally, if the repository is associated with a CVE record then the *CVE_ID* will contain the corresponding CVE identifier, otherwise it has a null value.

3.4.3 Commit Meta-data

AutoMESC collects commits from the collected OSS repositories for both Solidity and Vyper. Each commit has a unique hash and one commit is associated with a repository (*repo_id*) and may be associated with a CVE record. For each commit, AutoMESC extracts the *author*, *author_date*, *author_timezone*, *committer*, *committer_date*, *committer_timezone*, and Delta Maintainability Model metrics (*dmm_unit_size*) for code changes [3] meta-data. AutoMESC also extracts the message (*msg*) for further details of the commit and the fix action (i.e., adding or deleting lines of the vulnerability code).

3.4.4 Extracting Multiple Levels of Vulnerability-Fix Pairs

AutoMESC extracts vulnerability-fix pairs at two different levels of granularity - one based on files and one on lines. AutoMESC associates each commit with one or more *file_changes*, and each file change contains code diffs and the code of the file before (i.e., *code_before*) and after the change (*code_after*). Code diffs are presented in the same format delivered by Git.

3.4.5 Classification and Labeling of Extracted Vulnerabilities

AutoMESC classifies and labels vulnerable code using various classifications to assist with ML feature extraction models and other ML models. First, as mentioned earlier, it extracts the relevant CWE type from the corresponding CVE record and annotates the vulnerability accordingly with the extracted CWE

type. Then it extracts the description of each CWE from MITRE's[13] list of CWEs. Due to NVD's lack of distinction between CWE categories and CWE individual types, AutoMESC marks the field *is_category* with true. Furthermore, AutoMESC classifies the extracted vulnerabilities based on the CVSS, as discussed before. Finally, AutoMESC analyzes the contract code using 7 well-known tools, then classifies the file's vulnerability based on the results.

3.4.6 Dataset Exploration

Table 3.2 provides a statistical overview of the first release of the AutoMESC dataset. The total number of files in which there was a code change (i.e., a vulnerability or a fix) is 6.7K. A total of 10K contracts had vulnerability-fix changes.

Table 3.2: Statistics overview of the AutoMESC dataset

# GitHub Repo.	# Commits	# Files	# Contracts
4.4K	2.3K	6.7K	10K
# Methods	# Lines	CVEs	# of Vulnerabilities
6.8K	5.241K	0	97111

We also found out that CVEs have yet to be recorded for either Solidity or Vyper. According to this finding, it appears that the reported vulnerabilities on Ethereum smart contracts in the CVE records and NVD database exist in files other than (.vy) or (.sol) files. After manually analyzing the CVE records for Ethereum smart contracts, we found that most of the recorded vulnerabilities are interface-related. If future CVEs are recorded in Solidity or Vyper, AutoMESC will collect and store them in the database along with the related CWEs.

There are around 97K vulnerabilities in the first release of AutoMESC data. Among the most frequent vulnerabilities in AutoMESC data are hard-coded addresses, and implicit visibility levels, with more than 10K occurrences each. Table 3.3 shows the most frequently occurring labeled smart contract vulnerabilities in AutoMESC data. Finally, Table 3.4 shows that most of the vulnerabilities in AutoMESC have low severity levels, with 5.2% of the vulnerabilities having high severity.

[13] `https://cwe.mitre.org/data/index.html`

Table 3.3: Top 5 labeled smart contract vulnerability types using AutoMESC

Vulnerability Type	Total
Implicit visibility level	12536
Hardcoded address	11633
Upgrade code to Solidity	8431
Compiler version not fixed	7409
Comparison with block.timestamp	5116

Table 3.4: Severity levels distribution in AutoMESC data

Severity	1: (Low)	2: (Medium)	3: (High)
Percentage	85.3%	9.4%	5.2%

3.5 Evaluation

In this section, we evaluate the quality of the dataset and compare it against the related datasets discussed in Section 3.2.

To evaluate the dataset, we adopt the data quality taxonomy introduced by Bosu and Macdonell [5], which rates data quality issues according to three dimensions: accuracy, relevance, and provenance. Accuracy refers to the correctness of the data and is measured based on the following metrics.

- Incompleteness: missing data (md) (assigned as null, missing, or empty value). Incompleteness is measured by the percentage of missing labeled data in the dataset.
- Redundancy: duplicate data (dd) that are exactly the same. This is calculated by the percentage of the duplicated data.
- Inconsistency: contradicting or non-matching data (cd). We measure Inconsistency by the percentage of contradicting data.

The relevance class assesses the suitability of the data based on the following factors.

- Heterogeneity: diversity of the data source.

- Amount of data: the size of the dataset, including the number of attributes.
- Timeliness: the age of the dataset, and how regularly it is updated.

Provenance refers to the origin of the dataset. It is evaluated based on two metrics.

- Accessibility: data available to the public.
- Trustworthiness: the collection of data is documented and can be replicated.

Table 3.5 summarizes the general characteristics of the datasets and the quality evaluation results.

Table 3.5: Characteristics of related datasets and data quality evaluation results

Benchmark		[27]	[16]	[24]	[54]	Gigahorse benchmarks	Our dataset
General Charateristics	Purpose	Construct unbiased dataset	Evaluate SC tools	Construct unbiased dataset	Evaluate SC tools	Construct unbiased dataset	Construct unbiased dataset
Amount of data	Vulnerability Fixes	No	No	No	No	No	Yes
	Size	6.7K	143	46K	266	109	6.7K
	Size of labeled data	6.7K	143	564	266	109	6.7K
	# of Attributes	6	2	2	2	7	84
Heterogeneity	Source of data	Etherscan	Etherscan	Etherscan, SolidiFI, CVE, SWC and TCI	Literature-papers, Existing datasets	Etherscan	Github and CVE
	Supported vulnerability types	8	10	42	49	9	36
	Supported Languages	Solidity	Solidity	Solidity	Solidity	Solidity	Solidity and Vyper
	# of used tools	5	11	9	Manual	11	7
	CWE classification	No	No	Yes	No	No	Yes
Accuracy	Incompleteness	No	No evidence	No evidence	No evidence	0.92%	No
	Redundancy	No	No evidence	No evidence	No evidence	No	No
	Inconsistency	No evidence	No evidence	No evidence	No evidence	No evidence	No evidence
Timeliness		No Updates	No updates	No updates	No updates	No updates	Updates every 2 hour
Accessibility		Yes	Yes	Yes	Yes	Yes	Yes
Trustworthiness		Yes	Yes	Yes	Yes	Yes	Yes

3.5.1 Accuracy

For this class, we consider the datasets with an acceptable format such as CSV, JSON, etc. The datasets [16, 24, 54] are not evaluated for this class, as they are folder-based datasets, where the smart contracts are grouped in folders based on their vulnerabilities. These type of datasets cannot be used in data-driven approaches as the majority of them has only two attributes. Hence, in this class, we will only consider the Gigahorse benchmarks[14] and the Sujeet Yashavant et al. [27] dataset.

In terms of incompleteness, the Gigahorse benchmarks showed 0.92% missing data, while our dataset and Sujeet Yashavant et al. did not have any missing data. No redundancy was found in any of the datasets, even in the folder-based datasets. In all datasets, there was no evidence of any inconsistency issues. However, it was challenging to determine inconsistency in the datasets, as most of the datasets used different tools to detect vulnerabilities, and some of these tools might not have the same conclusion.

3.5.2 Relevance

In terms of heterogeneity, we have analyzed the source of data, the number of tools used to label the data, supported vulnerability types, supported languages, and if CWE classification is supported. All datasets are heterogeneous, where (i) the data were collected from at least two data sources, or (ii) they support multiple vulnerabilities. However, all the existing datasets only support the Solidity programming language, and not all of them support CWE classification.

The amount of data is determined based on the size of the dataset, the size of labeled data, and the number of attributes for each dataset. Some of these datasets are in the format of folders, where the attributes are the vulnerability type and the code file.

In terms of timeliness, the existing datasets are not updated after the data are made public. In our case, AutoMESC is designed so that the dataset can be updated every two hours.

3.5.3 Provenance

All the analyzed datasets are available publicly, and the collection of data is documented in detail as research papers or in GitHub instructions.

[14] `https://github.com/nevillegrech/gigahorse-benchmarks`

3.5.4 Evaluation Summary

Based on the above comparison, the AutoMESC dataset provides heterogeneous data with a variety of attributes that can be adapted for different data-driven research projects in the area of smart contract vulnerabilities. The dataset overcomes the current limitations in dataset timeliness where it updates the dataset every two hours based on newly disclosed vulnerabilities. This improves the quality of the AutoMESC dataset since smart contracts evolve rapidly, where new vulnerabilities are discovered and several vulnerability types become deprecated over time.

3.6 Threats to Validity

3.6.1 Internal Validity

A possible internal validity threat may result from an implementation bug in the codebase due to the complexity of the AutoMESC framework. We thoroughly tested the framework to address this concern. Moreover, we made the framework and the dataset publicly available so researchers and developers can verify the framework.

Each tool selected for vulnerability detection is neither sound nor complete. In some cases, a tool may generate a high number of false positives or false negatives. To mitigate this, we assessed whether a vulnerability exists or not based on the majority approach. However, such an approach could fail if the majority of the tools result in false positives or false negatives. These issues can be addressed either by integrating more tools or by manually verifying vulnerabilities through crowdsourcing or other methods. Nevertheless, these approaches require considerable time and resources, and we can apply them incrementally.

3.6.2 External Validity

A threat to external validity is the generalizability of AutoMESC. We built our dataset using publicly disclosed vulnerabilities and their fixes. These pairs may not reflect all vulnerabilities and fixes in smart contracts, particularly those not reported. To reduce this threat, we also collected vulnerability information from CVE and checked if these vulnerabilities are linked with Github repositories. If there is no match between collected vulnerabilities from CVE and Github repositories, we include these vulnerabilities in our dataset.

3.7 Conclusion

Several traditional and data-driven methods were proposed in the literature for improving the security of smart contracts, detecting their vulnerabilities, and perhaps fixing them. Nevertheless, data-driven research on smart contract vulnerabilities and fixes is still in its infancy, and a dataset with smart contract vulnerabilities and corresponding fixes is missing to support such research. This paper proposes a method of automatically mining and classifying smart contract vulnerabilities and their fixes under a fully automated framework called AutoMESC. AutoMESC constructs a dataset of vulnerabilities and fixes for smart contracts written in the most popular smart contract languages (Solidity and Vyper) mined from OSS projects hosted on GitHub and CVE records. The constructed dataset is enriched with meta-data that can support machine learning models for feature extraction. The initial dataset consists of approximately 6.7k smart contract vulnerability-fix code pairs with various levels of granularity and meta-data. In addition to opening up new opportunities for researchers in smart contract and empirical software engineering research, our framework can also be employed to identify smart contract vulnerabilities, automate their repair, and many more. In the future, we intend to empirically investigate the relationship between smart contract vulnerabilities and their fixes, and the patterns they show at different abstraction levels. Additionally, we plan to involve quantitative evaluation metrics to validate AutoMESC's effectiveness in mining and classifying vulnerabilities. Finally, we plan to explore the adaptation of the AutoMESC framework for just-in-time vulnerability detection in smart contracts [14].

Acknowledgment

This work was supported by the Icelandic Research Fund (Rannís) grant number 207156-051.

Bibliography

[1] Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). In: International conference on principles of security and trust, pp. 164–186. Springer (2017) 1, 53, 89, 126, 150, 194, 196, 215

[2] Bhandari, G., Naseer, A., Moonen, L.: Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In: Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering, pp. 30–39 (2021) 119, 121

[3] di Biase, M., Rastogi, A., Bruntink, M., van Deursen, A.: The delta maintainability model: Measuring maintainability of fine-grained code changes. In: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), pp. 113–122. IEEE (2019) 130

[4] Bosu, A., Carver, J.C., Hafiz, M., Hilley, P., Janni, D.: Identifying the characteristics of vulnerable code changes: An empirical study. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, pp. 257–268 (2014) 125

[5] Bosu, M.F., MacDonell, S.G.: A taxonomy of data quality challenges in empirical software engineering. In: 2013 22nd Australian Software Engineering Conference, pp. 97–106. IEEE (2013) 39, 132

[6] Buterin, V., et al.: A next-generation smart contract and decentralized application platform. white paper **3**(37) (2014) 1, 119, 191

[7] Choi, M.j., Jeong, S., Oh, H., Choo, J.: End-to-end prediction of buffer overruns from raw source code via neural memory networks. arXiv preprint arXiv:1703.02458 (2017) 119

[8] Coulter, R., Han, Q.L., Pan, L., Zhang, J., Xiang, Y.: Code analysis for intelligent cyber systems: A data-driven approach. Information sciences **524**, 46–58 (2020) 122

[9] Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P.: Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In: Proceedings of the ACM/IEEE 42nd International conference on software engineering, pp. 530–541 (2020) 14, 67, 119, 121, 122, 134, 135, 158, 196, 226

[10] Feist, J., Grieco, G., Groce, A.: Slither: a static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), pp. 8–15. IEEE (2019) 15, 128, 191, 195, 199, 224

[11] Ferreira, J.F., Cruz, P., Durieux, T., Abreu, R.: Smartbugs: A framework to analyze solidity smart contracts. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, pp. 1349–1352 (2020) 121

[12] Kolvart, M., Poola, M., Rull, A.: Smart contracts. In: The Future of Law and etechnologies, pp. 133–147. Springer (2016) 119

[13] Lin, G., Wen, S., Han, Q.L., Zhang, J., Xiang, Y.: Software vulnerability detection using deep neural networks: a survey. Proceedings of the IEEE **108**(10), 1825–1848 (2020) 122

[14] Lomio, F., Iannone, E., De Lucia, A., Palomba, F., Lenarduzzi, V.: Just-in-time software vulnerability detection: Are we there yet? Journal of Systems and Software **188**, 111283 (2022) 137

[15] Marchesi, L., Marchesi, M., Destefanis, G., Barabino, G., Tigano, D.: Design patterns for gas optimization in ethereum. In: 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE), pp. 9–15. IEEE (2020) 119

[16] Marina, M.: Smartembed code clone analysis of 200 000 smart contracts. URL `https://zenodo.org/record/6338506#.YmXcPOrP02w` 119

[17] Mehar, M.I., Shier, C.L., Giambattista, A., Gong, E., Fletcher, G., Sanayhie, R., Kim, H.M., Laskowski, M.: Understanding a revolutionary and flawed grand experiment in blockchain: the dao attack. Journal of Cases on Information Technology (JCIT) **21**(1), 19–32 (2019) 119, 145

[18] Mell, P., Scarfone, K., Romanosky, S., et al.: A complete guide to the common vulnerability scoring system version 2.0. In: Published by FIRST-forum of incident response and security teams, vol. 1, p. 23 (2007) 129

[19] Morrison, P., Herzig, K., Murphy, B., Williams, L.: Challenges with applying vulnerability prediction models. In: Proceedings of the 2015 Symposium and Bootcamp on the Science of Security, pp. 1–9 (2015) 122, 201

[20] Mueller, B.: Smashing ethereum smart contracts for fun and real profit. In: 9th Annual HITB Security Conference (HITBSecConf) (2018) 15, 128, 225

[21] Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th annual computer security applications conference, pp. 653–663 (2018) 15, 128, 225

[22] Protofire: Solhint. https://github.com/protofire/solhint 15, 128, 225

[23] Rameder, H.: Systematic review of ethereum smart contract security vulnerabilities, analysis methods and tools. Ph.D. thesis, Wien (2021) 53, 61, 64, 65, 75, 119

[24] Ren, M., Yin, Z., Ma, F., Xu, Z., Jiang, Y., Sun, C., Li, H., Cai, Y.: Empirical evaluation of smart contract testing: what is the best choice? In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 566–579 (2021) 14, 119, 121, 122, 134, 135

[25] Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., McConley, M.: Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE international conference on machine learning and applications (ICMLA), pp. 757–762. IEEE (2018) 122

[26] Soud, M., Liebel, G., Hamdaqa, M.: A fly in the ointment: An empirical study on the characteristics of ethereum smart contracts code weaknesses and vulnerabilities. arXiv preprint arXiv:2203.14850 (2022) 128, 156, 161, 179

[27] Sujeet Yashavant, C., Kumar, S., Karkare, A.: Scrawld: A dataset of real world ethereum smart contracts labelled with vulnerabilities. arXiv e-prints pp. arXiv–2202 (2022) 119, 121, 122, 134, 135

[28] Szabo, N.: Smart contracts: building blocks for digital markets. EXTROPY: The Journal of Transhumanist Thought,(16) **18**(2), 28 (1996) 119, 193

[29] Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y.: Smartcheck: Static analysis of ethereum smart contracts. In: 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), pp. 9–16. IEEE (2018) 15, 128, 191, 195, 196, 200, 224

[30] Torres, C.F., Schütte, J., et al.: Osiris: Hunting for integer bugs in ethereum smart contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference, pp. 664–676. ACM (2018) 15, 128, 195, 225

[31] Torres, C.F., Steichen, M., et al.: The art of the scam: Demystifying honeypots in ethereum smart contracts. In: 28th {USENIX} Security Symposium ({USENIX} Security 19), pp. 1591–1607 (2019) 128

[32] Vacca, A., Di Sorbo, A., Visaggio, C.A., Canfora, G.: A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges. Journal of Systems and Software **174**, 110891 (2021) 65, 119

[33] Vyperhub-io: Vyper smart contracts. URL `https://github.com/vyperhub-io/vyper-smart-contract` 123

[34] Zhang, P., Xiao, F., Luo, X.: A framework and dataset for bugs in ethereum smart contracts. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 139–150. IEEE (2020) 3, 13, 14, 15, 53, 54, 62, 63, 64, 65, 75, 90, 92, 119, 121, 122, 134, 135, 178, 199, 203, 205, 206, 213, 224

[35] Zheng, Z., Xie, S., Dai, H.N., Chen, W., Chen, X., Weng, J., Imran, M.: An overview on smart contracts: Challenges, advances and platforms. Future Generation Computer Systems **105**, 475–491 (2020) 53, 119

[36] Zou, D., Wang, S., Xu, S., Li, Z., Jin, H.: Vuldeepecker: A deep learning-based system for multiclass vulnerability detection. IEEE Transactions on Dependable and Secure Computing **18**(5), 2224–2236 (2019) 122

Chapter 4

Automated Prioritization of Smart Contract Vulnerabilities

PrAIoritize: Automated Early Prediction and Prioritization of Vulnerabilities in Smart Contracts

Soud, M., Liebel, G., and Hamdaqa, M. 2023.

In submission to a journal.

Abstract

Context: Smart contracts are prone to numerous security threats due to undisclosed vulnerabilities and code weaknesses. In Ethereum smart contracts, the challenges of timely addressing these code weaknesses highlight the critical need for automated early prediction and prioritization during the code review process. Efficient prioritization is crucial for smart contract security.
Objective: Toward this end, our research aims to provide an automated approach, PrAIoritize, for prioritizing and predicting critical code weaknesses in Ethereum smart contracts during the code review process.
Method: To do so, we collected smart contract code reviews sourced from Open Source Software (OSS) on GitHub and the Common Vulnerabilities and Exposures (CVE) database. Subsequently, we developed PrAIoritize, an innovative automated prioritization approach. PrAIoritize integrates advanced Large Language Models (LLMs) with sophisticated natural language processing (NLP) techniques. PrAIoritize automates code review labeling by employing a domain-specific lexicon of smart contract weaknesses and their impacts. Following this, feature engineering is conducted for code reviews, and a pre-trained DistilBERT model is utilized for priority classification. Finally, the model is trained and evaluated using code reviews of smart contracts.
Results: Our evaluation demonstrates significant improvement over state-of-the-art baselines and commonly used pre-trained models (e.g. T5) for similar classification tasks, with 4.82%-27.94% increase in F1-measure.
Conclusion: This paper introduces PrAIoritize, an automated approach that effectively prioritizes smart contract code weaknesses during the review process. By leveraging PrAIoritize, practitioners can efficiently prioritize smart contract code weaknesses, addressing critical code weaknesses promptly and reducing the time and effort required for manual triage.

4.1 Introduction

Smart contracts are Turing-complete programs operating on the blockchain to manage the flow of assets among the contractual parties [41]. Smart contracts shifted blockchain technology such as Ethereum from primarily storing and transferring cryptocurrencies (e.g., Bitcoin) to enabling a wide range of transactions and decentralized applications (DApps) in various fields [12]. Smart contracts are typically written in high-level programming languages such as Solidity [53].

Inheriting immutable, self-executing, and decentralized attributes from the underlying blockchain technology, smart contracts cannot be modified once deployed to the blockchain, and their execution is entirely contingent on their unchangeable code [53]. In Ethereum, modifying smart contracts to fix vulnerabilities can be costly and complex, involving deploying new versions of contracts and upgrading them [1]. These inherent characteristics ensure the reliability of smart contracts and set them apart from conventional software.

The distinguishing features of smart contracts make them vulnerable to security threats. For instance, Ethereum's unique gas system and smart contract immutability exacerbate existing security threats [8], e.g., often resulting in significant financial losses in case of an attack. Notable incidents such as the DAO attack, leading to the hack of about 50 million dollars, highlight the substantial risks associated with smart contract code weaknesses [43].

Challenges in smart contract maintenance extend beyond inherent code weaknesses. Automated smart contract security tools can help mitigating a significant number of attacks [7]. Still, the large number of audits and reviews generated by these tools presents a considerable challenge, particularly for auditors tasked with manual triage. Hence, there is a need for improved automation in triaging and prioritizing code weaknesses, towards building tools that are useful for practitioners [7]. Finally, the novelty of smart contracts and blockchain technology can result in a substantial amount of weaknesses that are unknown, thus increasing the urgency of prioritizing weaknesses and addressing critical ones.

Therefore, this paper aims to answer the following research question:

RQ1: *To what extent can smart contract code weaknesses be successfully prioritized during code review processes?*

To answer this question, we present an automated approach to prioritize smart

[1] https://docs.openzeppelin.com/learn/upgrading-smart-contracts

contract code weaknesses, called PrAIoritize. We start by quantifying the occurrence of zero-day vulnerabilities, vulnerabilities that are unknown prior to their disclosure, in Ethereum smart contract code by studying the timing of exploit disclosure in records of the Common Vulnerabilities and Exposures (CVE) database, thus providing a stronger motivation for our work and future work on automated triage in smart contract code. Then, we leverage a combination of Large Language Models (LLMs) and natural language processing techniques (NLP) to label unlabeled code weaknesses automatically and prioritize them. Smart contracts, being sequential code with dependencies among code statements, are similar to natural language text. Moreover, they often follow specific templates and standards, leading to recognizable patterns and repetitive codes and structures [10]. Therefore, LLMs and other NLP-based approaches have significant potential in automating the detection of code weaknesses in smart contracts and comment generation [23, 56]. We constructed a smart contract code weakness lexicon that includes various code weaknesses and corresponding impacts. Subsequently, we utilized this lexicon to automatically label unlabeled code reviews. We then applied the DistilBERT [38], to automatically prioritize smart contract code weaknesses. Moreover, we have made the models, scripts, and data used in this study openly accessible to encourage research replication and to facilitate further investigations by other researchers in the field.

Our evaluation results show that PrAIoritize outperforms two state-of-the-art baselines and four popular and well-known pre-trained models for text classification, with 4.82%-27.94% higher F1-measure, 2.35%-27.94% precision, and 3.57%-27.94% higher recall.

Paper Organization. We introduce the motivation in Section 4.2. In Section 4.3, we present the terminology and related background. Our methodology is detailed in Section 4.4. Our experimental evaluation is presented in Section 5.5. Section 5.6 presents the results and findings from our empirical analyses, followed by the discussion and implications in Section 4.7. Related work is discussed in Section 5.7. Potential threats to validity are examined in Section 5.8. Finally, the paper concludes in Section 5.9.

4.2 Motivation

In this section, we offer real-world examples to highlight the necessity of predicting and prioritizing smart contract code weaknesses in code reviews at an early stage. In real-world scenarios, developers face significant challenges when it comes to prioritizing code reviews. This difficulty arises from the need to

manually examine numerous code reviews submitted by developers [29]. Failure to prioritize these reviews effectively can result in wasted time and effort for developers [21]. For instance, reviewers may devote significant time meticulously analyzing or addressing a code review, only to discover that it is of low priority or remains unused within the smart contract code. It is a common practice to categorize code reviews with code weaknesses into different priority levels, such as critical, high, medium, or low. These levels are assigned by developers based on various factors, including the type and severity of the code weaknesses addressed, their impact, and other relevant considerations [54]. We elaborate on these priority levels for smart contract code reviews in Section 4.4 of our paper.

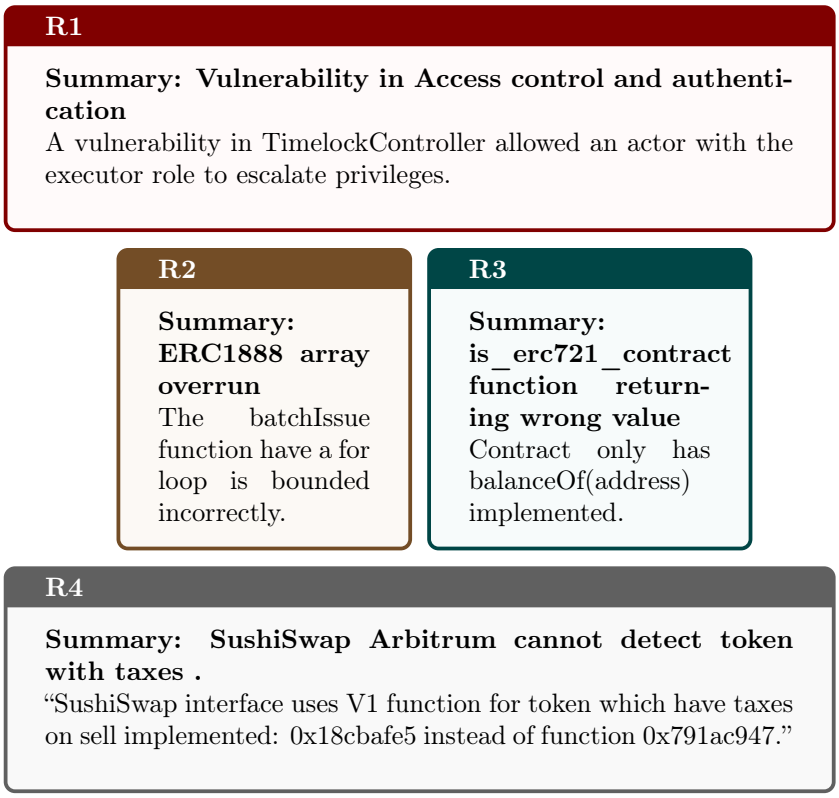


Figure 4.1: Illustrated examples: real-world smart contract code reviews

In Figure 4.1, we illustrate the four priority levels using examples of four real-world code reviews. In the first code review (R1) [27], a vulnerability is identified in the TimelockController. This vulnerability enabled an actor with the executor role to gain immediate control of the timelock by resetting the delay to 0 and escalating privileges. Consequently, the actor could obtain unrestricted access to assets stored in the contract. Instances with the executor role set to "open" posed a particular risk, as they allowed any user to exploit the executor role, thereby leaving the timelock vulnerable to takeover by potential attackers. If such a vulnerability were present in a high-value contract, such as the Axie Infinity[2] contract with a net value of $921 million [3], it could result in significant financial losses, client loss, and damage to reputation. Therefore, it is a critical priority code review, and it is crucial to fix the vulnerability to prevent attacks. Moreover, this vulnerability is reported and disclosed in the CVE record.

The second code review (R2) [25] demonstrates a code weakness in a function in the Registry contract. It is caused by an incorrectly bounded loop, which eventually may result in an array overrun or extra gas [4] consumption. While this specific weakness does not immediately expose the contract to exploitation or jeopardize control over its functionality (i.e., critical), it nonetheless demands high attention due to its impact on gas consumption within the contract. Excessive gas consumption in smart contracts not only results in the loss of Ether [5] but also leads to poor contract performance, potentially affecting user experience. On its own, the code weakness may not directly provide attackers with a means to compromise the contract's integrity or gain unauthorized access. Instead, it represents a potential avenue for exploitation when coupled with other vulnerabilities or when attackers leverage specific circumstances to exploit the weakness. Although deemed less critical than the vulnerability discussed in the first example, this weakness remains a high-priority weakness due to its significant potential impact on both the performance and security of the contract.

In the third review [26], a function is used to determine whether a contract adheres to ERC721 standard contract [6] or not. The code weakness is prompted when the function is invoked within the contract bytecode. If the code weakness is triggered, we consider it to be of medium priority as it can have notable

[2]Contract address: 0xbb0e17ef65f82ab018d8edd776e8dd940327b28b

[3]https://www.coingecko.com/en/coins/axie-infinity

[4]The unit for measuring the computational effort required to execute a transaction on Ethereum.

[5]The native cryptocurrency for the Ethereum blockchain.

[6]A widely adopted smart contract standard for representing ownership of unique non-fungible tokens [20].

implications on the contract interactions, especially those involving dependent contracts that rely on this information. Specifically, if triggered, the weakness may cause dependent contracts to incorrectly perceive the contract as non-compliant with ERC721 standards, potentially leading to cascading functional issues and possibly financial losses during interactions. This weakness, with its localized impact and lack of immediate exploitability, poses less imminent threats compared to previous examples that present more immediate risks, thus warranting higher priorities.

Finally, the last review (P4) [28] identifies a code weakness with the SushiSwap contract, one of the biggest decentralized exchanges (DEX), where the interface shows the wrong function for tokens with taxes, resulting in an error displayed in the console. The code weakness does not affect contract functionality or associated dependencies. Therefore, we consider this code review as a low-priority. Compared to the aforementioned code reviews, this code weakness is less critical and does not pose financial losses.

The definitions of the proposed different priority levels are provided in Section 4.4, while the limitations of this classification are listed in Section 5.8.

4.3 Background and Terminology

In this section, we provide essential background information and define key terms relevant to our research. We also provide background on smart contract security and zero-day attacks, highlighting the urgency of addressing vulnerabilities in smart contracts. We outline the primary objectives of code reviews and introduce pre-trained models utilized for automating the prioritization and prediction of code weaknesses in the existing literature.

4.3.1 Definitions

- **A vulnerability**: "A weakness in the computational logic (e.g., code) found in software and hardware components that, when exploited, leads to adverse effects on confidentiality, integrity, or availability. Addressing such weaknesses usually requires modifications to the codebase, but may also entail alterations to specifications or even the deprecation of certain specifications (e.g., eliminating affected protocols or functionalities entirely)" [46].
- **Weakness:** "a software error or mistake in contract code that in the right conditions can by itself or coupled with other weaknesses lead to a

vulnerability" [19]).

- **Priority**: Refers to the level of urgency assigned to a software code weakness, indicating how quickly the code weakness needs to be fixed and removed. Priority levels, such as high, low, medium, and critical, are commonly used to represent the urgency of addressing the weakness. However, the specific definitions of these priority levels may vary based on the field. The priority level is typically determined from the perspective of the software developers and is based on several factors, such as weakness severity, potential impact on system functionality, and business criticality [54].

- **Exploit:** A piece of software, or a sequence of commands that takes advantage of a code weakness, glitch, or vulnerability in order to cause unintended or unanticipated behavior to occur in computer software or hardware. It is commonly used to gain unauthorized access to a system, execute arbitrary code, or perform other malicious activities.

- **Third Party Advisory:** A notification or report issued by an external entity (e.g., a security researcher or a cybersecurity organization) regarding a security vulnerability or weakness found in a software product, system, or service. These advisories typically provide details about the vulnerability, its potential impact, and guidance on how to mitigate the risk.

- **Vendor Advisory:** A formal communication issued by the vendor or developer of a software product, system, or service to alert users about security vulnerabilities, bugs, or weaknesses identified in their products.

4.3.2 Smart Contract Security

Smart contracts, integral to blockchain technology, facilitate the decentralized execution of agreements and transactions without intermediaries. They have transformed various sectors by enhancing transparency, trust, and efficiency. Solidity, a programming language tailored for smart contracts, empowers developers to define their logic on platforms such as Ethereum. Despite the benefits that smart contracts provide, they are vulnerable to various security threats [48]. Furthermore, there have been numerous instances of smart contract security breaches in recent years [4]. The first attack occurred in 2016, when an attack on DAO contracts resulted in the loss of over 3.6 million Ethers due to

Table 4.1: Explanation of the collected CVE attributes

Attribute	**Explanation**
CVE ID	Unique identifier assigned to a CVE entry.
Publication Date	Date when the CVE entry was published.
Last Modified Date	Date when the CVE entry was last modified.
CVE Description	Description of the vulnerability or weakness described by the CVE entry.
Severity	Severity level assigned to the CVE entry, indicating the potential impact of the vulnerability.
CVSS2/CVSS3 Access Complexity	Complexity of access required to exploit the vulnerability.
CVSS2/CVSS3 Authentication	Authentication required to exploit the vulnerability.
CVSS2/CVSS3 Confidentiality	Impact on confidentiality if the vulnerability is exploited.
CVSS3 Attack Vector	Vector that describes how the vulnerability can be exploited.
CVSS3 Attack Complexity	Complexity of the attack required to exploit the vulnerability.
CVSS3 Integrity Impact	Impact on integrity if the vulnerability is exploited.
GitHub Link	Link to the GitHub repository or issue related to the CVE entry.
Exploit Date	Date when the vulnerability was exploited, if applicable.
Third Party Advisory Date	Date of a third-party advisory related to the CVE entry, if available.
Patch Date	Date when a patch or fix for the vulnerability was released, if available.

a re-entrancy vulnerability. In 2020, the security research team at CertiK discovered several vulnerabilities in the smart contract for the SushiSwap project [4] that allow the owner of the contract to perform unauthorized actions. Due to increasing attacks, smart contract security and trustworthiness have gained significant attention from scholars, resulting in numerous studies on smart contract vulnerabilities.

4.3.3 Common Vulnerability and Exposure (CVE) and National Vulnerability Database (NVD)

CVE (Common Vulnerabilities and Exposures) [7] is a catalog of publicly disclosed vulnerabilities and exposures managed by MITRE. It synchronizes with the NVD (National Vulnerability Database) [8], ensuring continuous alignment between the two. The NVD serves as a comprehensive repository containing information on all publicly known software vulnerabilities. It offers detailed insights into CVE-listed vulnerabilities, including exploit date, patch availability, and various search functionalities. Table 4.1 presents the attributes gathered from NVD for each CVE. It includes impact metrics such as the Common Vulnerability Scoring System (CVSS), vulnerability types categorized under the Common Weakness Enumeration (CWE), and other relevant metadata. Each CVE entry is assigned a unique identifier delineating the affected software product, sub-products, and different versions.

4.3.4 GitHub Code Reviews

A code review, also known as a peer review, is a systematic examination of software source code by one or more individuals other than the author(s) to identify defects, improve quality, and transfer knowledge. It is a fundamental practice in software development aimed at ensuring the reliability, maintainability, and security of the codebase [3]. Code reviews typically involve a team of developers collaboratively inspecting a piece of code line by line to identify code weaknesses such as vulnerabilities, defects, logical errors, performance bottlenecks, and violations of coding standards or best practices.

Code reviews offer several benefits, including early detection and resolution of code weaknesses, knowledge sharing among team members, and improvement of overall code quality.

[7] https://cve.mitre.org/
[8] https://nvd.nist.gov/vuln

A typical code review on GitHub encompasses various components, including a description of code weaknesses, a summary, auditor comments, code weakness details, dependency information, priority, severity, assignee, and proposed fixes. However, in the context of smart contracts, not all of these elements may be present. In our analysis, we focus on essential elements available in GitHub, particularly the Description field. These fields contain valuable information crucial for training our model and predicting the priority of code weaknesses.

4.3.5 Zero-Day Attacks

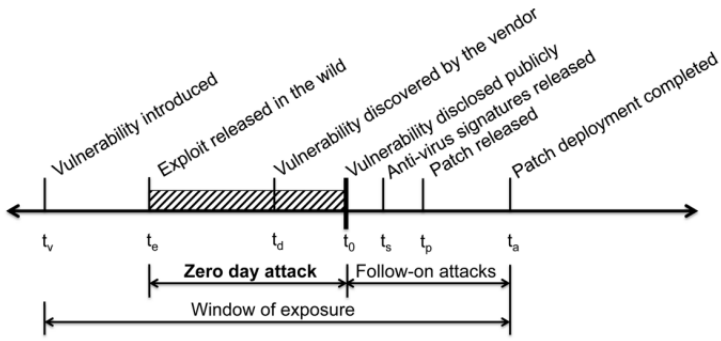


Figure 4.2: Timeline of zero-day attacks [6].

Zero-day attacks represent a class of cyber threats characterized by the exploitation of software or system vulnerabilities before vendors can develop and distribute patches or fixes. These attacks exploit vulnerabilities that are unknown to the vendor or for which no mitigating measures are available at the time of the attack. Consequently, zero-day attacks present a formidable challenge to organizations and individuals due to their ability to bypass existing security mechanisms without prior warning, potentially leading to data breaches, system compromises, and other security incidents with significant repercussions.

A zero-day attack timeline, as shown in Figure 4.2, typically begins with the introduction of a vulnerability into software ($time = tv$). Followed by the release of an exploit in the wild by malicious actors ($time = te$). Once the vendor becomes aware of the vulnerability, they assess its impact and begin working on a patch($time = td$). The vulnerability is then publicly disclosed and assigned a CVE identifier ($time = t0$). Then, anti-virus signatures are released

to detect ongoing attacks ($time = ts$). Finally, the vendor releases a patch ($time = tp$), and once it is deployed on all vulnerable hosts, the vulnerability ceases to have an impact ($time = ta$). A zero-day attack occurs when the vulnerability is exploited before it is publicly disclosed, i.e., $t0 > te$, highlighting the importance of timely patching. One of our goals in this paper is to measure the prevalence of zero-day vulnerabilities and assess the effectiveness of patching vulnerabilities before and after disclosure [6].

It is important to note that in smart contracts, the concept of releasing viruses is not applicable. Instead, when vulnerabilities are identified, fixes are released to address the code weakness and enhance security.

4.3.6 Related Models

Pre-trained language models have achieved remarkable success in various software engineering tasks [63, 62], such as code summarization and code weakness localization. This section briefly describes state-of-the-art pre-trained models that have been used to perform classification tasks similar to our task and achieved high performance.

Bidirectional Encoder Representations from Transformers (BERT) is a pre-trained language model proposed by Google [14] that has achieved state-of-the-art performance on different software engineering tasks (e.g. [13]). BERT can learn dynamic context word vectors and capture textual semantic features.

DistilBERT [38] is designed to be more memory efficient and faster to train than BERT.

T5 (Text-To-Text Transfer Transformer) is a language model developed by Google AI Language. It belongs to the Transformer architecture family [58]. T5 frames all tasks as text-to-text transformations [36]. This means that T5 is trained to input a text prompt and generate the corresponding output text. Its applications include various tasks such as translation, summarization, question answering, and text generation, as text transformation problems.

Bidirectional Long Short-Term Memory (BiLSTM) is a popular neural network architecture widely used for natural language processing tasks, including text classification [39]. As BiLSTM processes the input sequence in both directions, it can capture past and future contexts.

Recurrent Neural Networks (RNNs) are commonly used for text classification tasks [37]. RNNs are designed to effectively capture input text sequential dependencies.

4.4 Methodology

In this section, we outline the approach we took to investigate zero-day vulnerabilities in smart contracts. Next, we detail the different stages of our automated prioritization method, PrAIoritize.

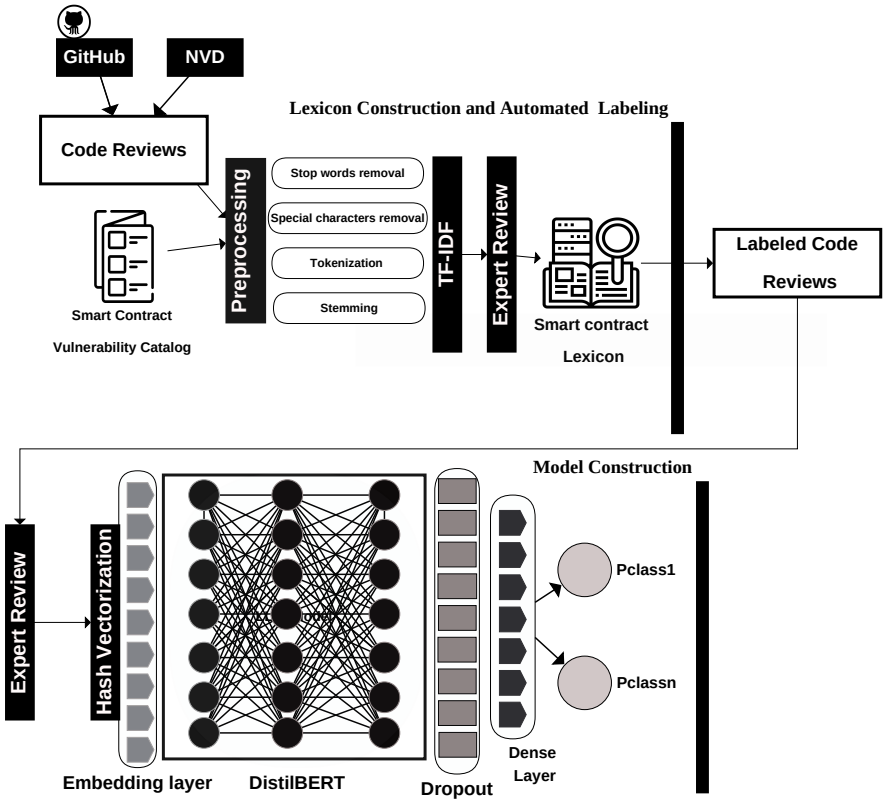


Figure 4.3: PrAIoritize approach overview

4.4.1 Definition of Priority Levels for Smart Contract Weaknesses

In our study, we define the priority levels for smart contract code weaknesses based on an analysis we conducted in previous research and our findings [52]. Specifically, we analyzed code weaknesses in smart contracts, drawing insights from our examination of various sources, including Stack Overflow, Github, CVE, and other relevant literature. Through this process, we identified prevalent code weaknesses and corresponding impacts in smart contract code. Building on these insights, we define the four priority levels in smart contracts, taking into account several factors. First and foremost is **exploitability**, assessing the extent to which attackers could capitalize on a weakness to compromise the contract integrity. **Impact** also plays a crucial role, in analyzing the extent of potential harm, whether it is financial losses or performance issues. Additionally, the **locality** of effects is evaluated to determine whether a weakness's impact is confined to specific components or has broader implications. By considering these factors, we define four levels of code weakness priority as follows:

- **Critical priority**: The code weakness present in the smart contract code. It poses an immediate risk to the contract's integrity, security, and functionality, potentially resulting in the loss of access to the entire contract. These weaknesses are highly exploitable by attackers and result in significant financial losses such as access control weaknesses. Critical priority weaknesses demand urgent attention and mitigation due to their high exploitability and severe impact on the contract's operation and security.

- **High priority**: The code weakness in the smart contract causes significant undesirable outcomes. In contrast to a critical weakness, it cannot be exploited directly by attackers. High-priority weaknesses may be coupled with existing vulnerabilities, leading to cascading failures or performance degradation.

- **Medium priority**: The code weakness is within the contract implementation and may result in unintended logic. While not directly exploitable by external attackers, these weaknesses can affect dependent contracts and may lead to incorrect functionality when interacting with other decentralized applications (DApps). Such weaknesses have the potential to disrupt the seamless operation of interconnected DApps and impact the overall interoperability of the smart contract ecosystem.

- **Low priority**: The code weakness does not affect the contract's functionality or any associated environment calls outside the contract. These code weaknesses may be related to unused variables in the contract code, errors in the contract's interface, documentation, or other non-essential components.

4.4.2 Overall Approach

Our approach starts with the data collection phase, where we collect CVEs from the NVD database and code reviews from GitHub related to Ethereum smart contracts. We then conduct an investigation into zero-day vulnerabilities by quantitatively analyzing CVE records, including exploit dates, third-party advisory dates, vendor advisories, CVE patches, and common weaknesses (CWE). Motivated by the findings from our quantitative analysis, we propose PrAIoritize. PrAIoritize leverages LLMs and NLP to predict the priority levels of smart contract weaknesses. As shown in Figure 4.3, PrAIoritize consists of three main phases: (1) lexicon construction and automated labeling phase, (2) classification model construction, and (3) training phase.

The first phase of our methodology involves collecting well-known smart contract code weaknesses and vulnerabilities in a smart contract vulnerability catalog, then using the vulnerability catalog along with the code reviews collected from GitHub and CVE to construct a lexicon of code weaknesses associated with impacts in smart contracts, as well as their respective priority levels. Followed by automatically generating priority labels for unlabeled code reviews using the constructed lexicon. In the model construction, PrAIoritize utilizes feature engineering to capture textual and semantic factors that may impact the priority level of a code weakness. These features are then passed through a classification model to create a model capable of classifying a code review with an unknown priority level. Finally, in the prediction phase, PrAIoritize leverages the DistilBERT model to predict the priority levels of a given set of code reviews. The features of each code review are extracted, including tokenized text, embeddings, semantics, and other relevant characteristics, and are utilized by the model to analyze and predict the priority level. We further detail our approach in the following sections

4.4.3 Dataset Collection

To answer the proposed research question, we utilized AutoMESC [43] to extract data from two well-known publicly accessible sources; CVE from NVD

and GitHub. These sources are commonly used for reporting vulnerabilities in Ethereum smart contracts and other software systems. Table 4.2 describes the data sources and the final selected number of reviews per source after pre-processing.

Data-source 1— CVE. We utilized AutoMESC to collect and extract CVEs associated with smart contracts between 2018 and 2023 from the NVD database. The collected data include various attributes and metadata, such as CVE ID, publication date, last modified date, CVE description, severity, CVSS2 access complexity, CVSS2 authentication, CVSS2 confidentiality impact, CVSS3 attack vector, CVSS3 attack complexity, CVSS3 integrity impact, GitHub link, exploit date, third-party advisory date, and patch date. Table 4.1 identifies the collected attributes per CVE as described in the NVD website.

Data-source 2— GitHub. We opted to utilize GitHub as our secondary data source for this study due to its status as the most widely used social coding platform, as highlighted in previous research [15]. Additionally, numerous studies exploring Ethereum smart contracts and related analysis tools have relied on data sourced from GitHub's open-source projects [16]. Our data collection concentrated on identifying code weaknesses across various open-source projects featuring Ethereum smart contracts written in Solidity[9].

We initially collected relevant code reviews corresponding to the collected CVEs from Data-source1 utilizing the provided Github Link for each CVE. Next, we employed targeted keywords such as "smart contract", "Solidity", and "Ethereum". At the same time, we conducted searches using terms such as "issue", "defect", "weakness", "problem", and "vulnerability" to broaden our collection of issues related to Ethereum smart contracts. Additionally, we included the Ethereum official GitHub repository in our analysis.

Then, we excluded code reviews with open issues in smart contracts and concentrated our efforts on those with closed issues. This decision was made to ensure a focused examination of verified weaknesses and maintain the integrity of our research findings. Our focus was specifically on newly published code reviews within the four years, up until February 2023, to capture the most recent and relevant information. For noise reduction, code reviews with fewer than 5 words were excluded. The detailed number of code reviews per priority level is also listed in Table 4.2.

Furthermore, the dataset is automatically and randomly divided into training and testing sets, with an 80/20 ratio, respectively.

[9]All the smart contracts and the issues are listed in our artifact

Table 4.2: Summary of sampled smart contract reviews with code weaknesses in selected data sources

Data Source	GitHub	NVD	Total
# of collected data	2100	442	2542
# of data records after preprocessing	1000	440	1440

4.4.4 Investigating Zero-Day Vulnerabilities in Smart Contracts

To identify zero-day vulnerabilities, we leveraged the attributes collected from *Data-source 1* for each CVE. We focused on extracting key dates, including the exploit date, the third-party advisory date, the vendor advisory date, and the patch date whenever available. Moreover, we conducted an in-depth analysis of the temporal relationships among these dates. Precisely, our analysis involved examining the proximity between the exploit date and the other dates (i.e., the third-party advisory, vendor advisory, and patch dates) following the timeline discussed in the background section 4.3. This analysis enabled us to pinpoint the vulnerabilities that indicate zero-day attack scenarios.

The analysis was done automatically, by retrieving input data from the specified source and extracting key dates. If an exploit date is absent, but an advisory date exists, indicating no attack, the analysis proceeds with the next CVE record. However, if both exploit and advisory dates are identified, the temporal difference is analyzed. Instances where the exploit date precedes or coincides with the advisory date signify zero-day vulnerabilities. Furthermore, our automatic analysis detects instances where fixes were implemented before attacks occurred, indicating preemptive patching of vulnerabilities. Ultimately, statistics on zero-day vulnerabilities within the dataset are calculated, offering insights into the prevalence of such vulnerabilities in smart contracts.

4.4.5 Data Cleaning and Preprocessing

In this phase, each code review is preprocessed and cleaned to ensure that it does not have any data quality issues that may negatively affect the classification and prediction results. Code reviews consist of textual descriptions and code snippets, and it is crucial to process the textual description with care, as even a small error can lead to the exclusion of useful information.

Table 4.3: Statistics of collected code weaknesses per priority level

Code weaknesses per Priority Level				
	Low	**Medium**	**High**	**Critical**
Total	192	179	234	395

We first extracted the code weakness description from each code review. Then, we removed all special characters and punctuation characters from each code review. These elements have no significant effect on the text mining process, and removing them from the text data ensures optimal performance of the text mining algorithms [5].

We then manually remove the URLs, code snippets, configurations, and transaction logs to ensure that the model can better comprehend the textual description. This is done to minimize the potential for the model to be confused by extraneous information and to better focus on the main content of the code review. By simplifying the code review in this way, we aim to improve the model's ability to accurately predict the priority level of the code weaknesses. We outline the primary techniques that we employ to preprocess in the following paragraphs.

Tokenization: we utilized tokenization to break up a continuous stream of text into individual units called tokens. In the case of preprocessing the code reviews, the textual description is tokenized, meaning it is broken down into individual tokens, each of which refers to a word in the description. The extracted words are separated by delimiters, which in our case are white spaces.

Stop Words Removal: These are commonly used words that may not carry much meaning in the context of the code reviews, such as pronouns. We utilized a pre-existing list [40] of stop words to identify and remove them from the extracted corpus of code reviews.

Stemming: In the English language, words have multiple forms, making analysis challenging. We used stemming to resolve this problem by converting words to their root form. Stemming involves reducing words to their base form, (i.e., "attack" is the base form of "attacking", "attacked", and "attack"), which aids in analyzing the meaning of the text.

4.4.6 Automated Labeling

Due to the increasing complexity of smart contracts, the need for an automated labeling approach that can efficiently and accurately prioritize code reviews has become more urgent. Automated labeling has been used in various Natural Language Processing (NLP) tasks in the literature [38], such as sentiment analysis [38, 35] and text summarization [23].

In this study, we propose lexicon-based automatic labeling to assign priority labels to unlabeled code reviews.

Lexicon Construction. One of the essential steps in our approach is constructing a lexicon of smart contract vulnerability-related keywords along with their priority levels. The proposed lexicon consists of keywords, and each keyword has a priority level. Lexicon construction process consists of several steps as shown in Figure 4.3 and Algorithm 1. First, a vulnerability catalog is assembled by collecting well-known smart contract vulnerabilities and code weaknesses from the literature [52] and from DASP[10] which is a popular source of the top 10 critical security risks in smart contracts to include them in the lexicon. After that, the vulnerability catalog and the corpus of code reviews from the NVD and GitHub are preprocessed and concatenated using the techniques discussed in section 4.4.5 as shown in Algorithm 1-step 1. Then, index term extraction and term-weighting are applied using the term frequency-inverse document frequency (TF-IDF) weighting algorithm [16] and Bag of Word (BoW) [50] to extract keywords from the textual descriptions of the preprocessed corpus of code reviews and vulnerability catalog (i.e., Algorithm 1-step 2). This step includes assigning each term in the textual code review with a numerical score using the TF-IDF weighting scheme that ranks the terms based on both the frequency of the word in each code review (term frequency) and the rarity of the word in the entire corpus (inverse document frequency). In addition to representing the textual descriptions of the code reviews as a bag (i.e., multiset) of its words while considering the frequency of each word in the review (i.e., Algorithm 1-step 3). The resulting keywords are sorted based on their term frequency. Then, an expert—specifically, a doctoral student specializing in software engineering with a focus on smart contract security and software security [11]—assigned priority levels to the top 250 keywords. This assignment takes into account their respective impacts as identified in the code reviews. The lexicon is constructed based on the resulting keywords with their priority levels. The lexicon can enable efficient automatic labeling of smart contract reviews.

[10] https://dasp.co/

[11] The first author of this paper.

After employing a randomization script, the expert meticulously reviewed the keywords and their assigned priorities after the construction phase, following an iterative expert review process similar to established practices in the field [9, 7], to ensure the correctness of the lexicon.

The use of both TF-IDF weighting scheme and the BoW for pre-processing the lexicon-based approaches is a well-established technique in information retrieval [42, 50]. In the following, we describe the details of the TF-IDF used in our approach.

Algorithm 1: Smart-Contract Vulnerability-related Keywords Lexicon Construction

Input : (i) RG: review with code weaknesses from GitHub (ii) RC: review with code weaknesses from NVD (iii) KT: extracted key terms (iv) TFIDF: term frequency function used to create sorted words (v) Rj: a code review

Output: Lexicon with index numbers generated from TFIDF

STEP 1: Preprocess and concatenate RG and RC to create a corpus C

STEP 2: Extract KT from C to create a list of key terms T

STEP 3: **for** *each* Rj *in* C **do**
  if Rj *has been previously processed* **then**
    Continue
  end
  for *each term* Ti *in* Rj **do**
    $F1 \leftarrow TFIDF(term(Ti), Rj)$
  end
end

STEP 4: Save all the results in a file

STEP 5: Sort the words in B based on their term frequency in descending order in F1

STEP 6: Select the top 250 terms and assign index numbers to the sorted words to create the lexicon

STEP 7: Return the lexicon

Details of the TF·IDF and BoW. We use *TF-IDF* function (i.e., Algorithm 1-step 3) to apply the term-weighting algorithm [50] on the corpus of the code reviews. Let C denote a corpus of code reviews and $Rj_c \in C$ denote a code review in the corpus. The term frequency, tf(T_i, Rj_c), is defined as the number of times a term T_i occurs in the code review Rj_c. Let C_{T_i} denote the set of code

reviews in the corpus C that contain the term T_i. The number of code reviews in which the search term T_i appears is denoted as $|C_{T_i}|$, while $|C|$ represents the total number of code reviews in the corpus C. The inverse document frequency (idf) of term T_i in corpus C is defined as $\text{idf}(T_i) = \log \frac{|C|+1}{|C_{T_i}|+1}$.Moreover, the addition of '+1' in the denominator, specifically '$|C| + 1$', prevents division by zero and addresses the scenario where a term occurs in every document in the corpus ($|C_{T_i}| = |C|$).

The TF-IDF weight of a term T_i is proportional to the number of times it appears in a code review, Rj_c, and inversely proportional to the number of code reviews where the term occurs. In order to calculate TF-IDF, we used the following formula.

$$\text{TF} \cdot \text{IDF}(T_i, Rj_c) = \text{tf}(T_i, Rj_c) \times \log \frac{|C| + 1}{|C_{T_i}| + 1} \tag{4.1}$$

As shown in Formula 4.1, TF-IDF is the combination of TF and IDF functions. When a term occurs in a document frequently, its TF-IDF weight increases, and when it occurs in a corpus frequently, it decreases. For instance, stop words are common terms that occur in a large fraction of textual descriptions of the code reviews and do not contribute much to discriminating them. *Therefore, the idf part in TF-IDF helps to measure the importance of a term in the code review. If a term appears frequently in the code review, it may not be very helpful in identifying relevant reviews. The idf helps to downplay the significance of such terms.* Moreover, we apply BoW function in the same way as TF-IDF. However, we use the following function.

$$\text{BG}(T, Rj) = \sum_{i=1}^{n} [t_i = T] \tag{4.2}$$

Where BG is the BoW function, $\sum_{i=1}^{n} [t_i = T]$ is the count of occurrences of the term T in the code review Rj. It is important to note that BG function performs similarly to TF-IDF and represents the count of occurrence of t_i in R_j.

4.4.7 Model Construction

We constructed a model for code review classification in smart contracts using a pre-trained DistilBERT model, a variant of BERT (Bidirectional Encoder Representations from Transformers), which has been distilled for faster and

more efficient processing while retaining much of BERT's performance. The model architecture leverages the Transformer architecture, which has shown remarkable success in NLP tasks [38].

The model consists of an input layer, a DistilBERT layer, and a dropout layer with a dropout rate of 0.2 to prevent overfitting, followed by two dense layers with 128 and 64 hidden units, respectively. Dropout regularization is applied after the DistilBERT layer to prevent overfitting by randomly setting a fraction of input units to zero during training. The output layer, a fully connected dense layer with a softmax activation function, predicts the probability distribution over the classes.

Input process layer In this layer, we utilize the HashingVectorizer [34] because it provides a computationally efficient method for preprocessing and transforming text data into fixed-size numerical vectors. It can effectively handle out-of-vocabulary words [60], as it does not require a pre-built vocabulary. This attribute makes it more robust when encountering new, unseen words that might not have been present in the training data. HashingVectorizer performs well with heterogeneous data [24]. However, HashingVectorizer may not capture semantic relationships between words as effectively as other sophisticated embedding techniques such as Word2Vec [45] or GloVe [47]. In this layer, we initialize the HashingVectorizer with a fixed number of features (10,000). It is then used to transform the training and validation texts into numerical vectors. By setting the number of features to 10,000, the HashingVectorizer creates a 10,000-dimensional vector representation for each input text. The HashingVectorizer layer applies a hash function to the individual tokens (words) in the code review, mapping these hashed values to a fixed-size vector space. The vector representation is created by summing the hashed values for each token in the code review, resulting in a fixed-size vector representation for each code review, regardless of its original length. The hashing function can be represented by the following formula:

$$h(t) = t \bmod N \tag{4.3}$$

where $h(t)$ is the hash function, t is the input token, and N is the number of features (in our case, 10,000). The hashing function maps the input token to a position in the vector space by taking the remainder of the division of t by N.

Output Layer (Dense, Softmax Activation): The output layer consists of a dense layer with the number of units equal to the number of priority levels, which is 4 in our case. The softmax activation function is applied to the output layer for multi-class classification. This activation function converts the raw output scores from the neural network into probabilities, ensuring that the pre-

dicted probabilities for each class sum up to 1. The softmax function computes the probability $P(\text{class}_k)$ for each class k using the following formulas:

$$Z_i = \exp(a_i) \quad \text{for } i \in \{1, 2, \ldots, K\} \tag{4.4}$$

In Equation 4.4, a_i represents the raw output score for the i-th class produced by the neural network, and K is the total number of classes ($K = 4$ in our case). The exponential function $\exp(\cdot)$ is applied to each a_i to obtain positive scores Z_i for every class.

$$P(\text{class}_k) = \frac{Z_k}{\sum_{i=1}^{K} Z_i} \quad \text{for } i \in \{1, 2, \ldots, K\} \tag{4.5}$$

In Equation 4.5, $P(\text{class}_k)$ represents the probability of the input belonging to the k-th class. To calculate this probability, the exponential score Z_k for a class k is divided by the sum of the exponential scores for all classes. This normalization step ensures that the probabilities of all classes sum up to 1.

The model architecture includes a dropout layer with a dropout rate 0.2 to prevent overfitting. This dropout layer is followed by two dense layers with 128 and 64 hidden units, respectively. Dropout regularization is applied after the DistilBERT layer to prevent overfitting by randomly setting a fraction of input units to zero during training. The output layer, a fully connected dense layer with a softmax activation function, predicts the probability distribution over the classes.

4.4.8 Training Details

In this work, we fine-tuned the hyperparameters of PrAIoritize to achieve optimal performance. The tuning process was conducted during the training phase, with hyperparameters adjusted based on the performance of the model on the validation set. We utilized the Sparse Categorical Crossentropy loss function, which is suitable for multi-class classification tasks. Additionally, we employed Sparse Categorical Accuracy as the evaluation metric to measure the model's performance.

The training process involved iterating over the training dataset for a total of 10 epochs, with a batch size of 64 for each training iteration. Each epoch consisted of a training loop and an evaluation loop. During the training loop, the model parameters were updated using backpropagation to minimize the loss function. Dropout regularization with a dropout rate of 0.2 was applied as mentioned earlier.

During the training process, we monitored the loss and accuracy metrics on both the training and validation datasets. At the end of each epoch, we evaluated the model's performance on the validation set using classification metrics. We also generated a confusion matrix on the validation set to visualize the distribution of predicted labels compared to the ground truth labels. We employed Adam optimization algorithm with default settings for the learning rate and other parameters. The descriptions of the reviews are first preprocessed using a HashingVectorizer with 10,000 features, followed by one-hot encoding of the labels. This prepares the data for input into PrAIoritize. We implement PrAIoritize using the open-source Keras library[12] built on top of TensorFlow [1]. PrAIoritize is trained on the training dataset and evaluated on the validation dataset. The best model weights achieved by PrAIoritize are saved during training using a checkpoint callback based on validation accuracy. Finally, we assessed PrAIoritize performance by utilizing classification reviews and evaluation matrices.

4.5 Experiment Evaluation

In this section, we describe our experiment and evaluation measures. Moreover, we present our findings.

4.5.1 Baseline Selection

Baseline 1: We utilize the approach proposed by Meng et al. [44] in which the text information of the code weakness along with the weakness explanation was used to classify code weakness reviews. The proposed approach utilizes BERT and TF-IDF to extract the features of the text information, and then it trains machine learning classifiers (e.g. K-Nearest Neighbor) to classify the code weaknesses. We consider the textual features part and implement the model strictly as described in the paper, as the code is not provided in the paper.

Baseline 2: We also considered DRONE as it is the most cited state-of-the-art approach by Tian et al.[54]. Drone proposed GRAY (ThresholdinG and Linear Regression to ClAssifY Imbalanced Data) to classify code weaknesses based on several features extracted from six dimensions, i.e., textual, temporal, author, related report, severity, and product. Because smart contracts are still in their early stages, most of these dimensions are not available in the code

[12]https://github.com/keras-team/keras

reviews yet, so we consider the textual dimension only. We implement the model precisely based on the description provided in the research paper, as the source code is not available.

4.5.2 Evaluation Measures

To evaluate the performance of PrAIoritize, we use commonly used metrics in the literature, namely precision, recall, and F1-measure.

Precision represents the ratio of correctly predicted instances to the total number of predictions. Recall, on the other hand, is the ratio of correctly predicted instances to the total number of actual instances. F1-measure is a harmonic mean of precision and recall, and it provides a balanced evaluation of the model's performance. In the following, we summarize the evaluation metrics used in our analysis:

$$\text{Accuracy: } \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}} \tag{4.6}$$

$$\text{Precision: } \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{4.7}$$

$$\text{Recall: } \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{4.8}$$

$$\text{F1-measure: } 2 * \frac{\text{Precision * Recall}}{\text{Precision} + \text{Recall}} \tag{4.9}$$

Where:

- True Positive (TP): The model correctly predicts a positive sample as positive.
- True Negative (TN): The model correctly predicts a negative sample as negative.
- False Positive (FP): The model incorrectly predicts a negative sample as positive.
- False Negative (FN): The model incorrectly predicts a positive sample as negative.

4.6 Empirical Results

In this section, we answer the proposed RQ and list our results and findings.

4.6.1 Zero-Day Vulnerabilities in Smart Contracts

Using the automatic analysis outlined in the methodology section 4.4, we identified zero-day vulnerabilities in smart contracts. In our analysis, we excluded four CVEs (i.e., CVE-2020-17752, CVE-2018-17882, CVE-2018-18665, and CVE-2018-18666) due to unclear disclosure dates in third-party advisories, opting to omit them from the analysis.

The analysis of vulnerabilities within smart contracts underscores the significant risk posed by zero-day attacks. The findings reveal that a substantial portion, approximately 77.22%, of identified vulnerabilities were exploited by attackers before any mitigation measures could be implemented. This category represents vulnerabilities that were leveraged without prior disclosure or available fixes. Additionally, approximately 20.27% of vulnerabilities were identified but not actively exploited, suggesting a potential window of opportunity for auditing teams to address these weaknesses before they are exploited. Conversely, a smaller fraction of vulnerabilities, accounting for 1.14%, were patched by developers before any exploitation occurred. Nevertheless, a portion of vulnerabilities, approximately 1.37%, remains of uncertain exploitation status as the dates were not clearly listed in the vulnerability information as mentioned earlier.

Furthermore, we analyzed the sources of code reviews associated with each CVE. Our investigation revealed that these reviews primarily originate from three main sources: GitHub, OpenZeppelin forums, and Ethereum forums exclusively. Notably, GitHub emerged as the most prevalent source of code reviews across the CVEs analyzed. In contrast to traditional software, where CVEs may utilize specified prioritization systems, for instance, as seen in CVE-2018-13093 and CVE-2018-13095, the nature of smart contract vulnerabilities and their associated code reviews demonstrates a distinct reliance on community-driven platforms for vulnerability disclosure, prioritization, and patching.

Summary

- Among the analyzed smart contract vulnerabilities, 77.22% were identified as zero-day vulnerabilities, indicating a significant prevalence of previously unknown vulnerabilities exploited by threat actors.
- Smart contract vulnerabilities lack dedicated tracking systems. Instead, the disclosure, prioritization, and patching of vulnerabilities rely heavily on community-driven platforms and associated code reviews.

4.6.2 Answers to RQ1: Prioritization of Smart Contract Code Weaknesses

In order to evaluate the performance of the proposed automated labeling, an expert manually reviewed the labeled code reviews that were obtained by automatic labeling. In the manual review, the expert removed all the labels generated by the automated labeling and labeled the data according to their priority level. The manual review is based on the terminology proposed in the Background section 4.3. Then, we calculated the inter-rater agreement using the kappa coefficient [59] between the data labeled by the expert and the automatically generated labels. The resulting kappa coefficient is 0.92, which indicates a high level of agreement. We observed that most disagreement cases occurred within the critical and high-priority levels, revealing a tendency for ambiguity between these priorities in our analysis. This ambiguity suggests a lack of clear distinction in documentation between high and critical priorities. Additionally, our manual labeling process revealed inadequate descriptive terms for high-priority reviews, where terms such as 'weaknesses', 'issue' and 'vulnerability' were sometimes used interchangeably, leading to confusion. Furthermore, these reviews often lack sufficient information about the exploitability or potential impact of identified weaknesses. The distinction between high and critical-priority reviews lies in the severity of the vulnerabilities they expose and their potential for exploitation. Critical-priority reviews identify vulnerabilities with significant impact, allowing attackers unauthorized access to the smart contract and compromising its integrity and security. Conversely, high-priority reviews highlight weaknesses that, while impactful, may not immediately grant unauthorized access but still represent substantial weaknesses within the contract itself, affecting

Table 4.4: PrAIoritize performance

Priority Level	Precision	Recall	F1-measure	Average
Low	0.81	0.86	0.83	0.833
Medium	0.81	0.83	0.82	0.82
High	0.92	0.80	0.86	0.86
Critical	0.95	0.99	0.97	0.97

contract performance. However, these weaknesses are often poorly described in the reviews, contributing to underestimation of their impact and overlooking significant flaws. These differences extend beyond mere vulnerability identification, influencing both the impact and potential exploitability of identified weaknesses

Overall, our review shows that the proposed automated labeling performs similarly to the manual labeling in terms of inter-rater agreement. This finding supports the usefulness and validity of automatically prioritizing code reviews in smart contracts as a first step in the labeling process to reduce time and effort.

As shown in Table 4.4, PrAIoritize predicts the four priority levels with an average F1-measure of 83%, 82%, 86%, and 97%, respectively. The F1-measure for the critical level is the best and the medium priority level is the lowest. Low and medium levels are close in F1-measure. Taking the mean of the F1-measure for all priority levels, regardless of their support, the macro average of F1-measure is 87%, indicating reasonably good performance in predicting priority levels. Nevertheless, we believe it is highly important for code review prioritization to have higher accuracy and F1-measure for the critical level than other priority levels, which PrAIoritize is able to provide. Moreover, the consistency of F1-measure values across different priority levels highlights the reliability of the PrAIoritize model. The consistent performance across the four priority levels suggests the model's adaptability and effectiveness in handling various code review complexities and priorities.

Moreover, we investigate how effective frequently used pre-trained models are as well as the selected baselines, namely DRONE and Meng et al. [44] for the same classification task. Table 4.5 shows the F1-measure of PrAIoritize versus the two baselines and four popular pre-trained models for text classification tasks (i.e., BERT, T5, BiLSTM, and RNN). We utilized identical parameters and configurations as those detailed in the PrAIoritize method section 4.4. We

Table 4.5: Comparisons of F1-measure of PrAIoritize versus other classifiers and baselines

Priority Level	**Low**	**Medium**	**High**	**Critical**	**Average**
PrAIoritize	0.83	0.82	0.86	0.97	**0.87**
[44]	0.63	0.38	0.73	0.96	**0.68**
DRONE	0.29	0.10	0.32	0.47	**0.30**
BERT	0.75	0.80	0.82	0.97	**0.83**
T5	0.25	0.10	0.46	0.92	**0.43**
BiLSTM	0.79	0.77	0.66	0.95	**0.79**
RNN	0.76	0.70	0.52	0.95	**0.73**

also briefly describe the selected models in the Background section 4.3. We notice that Meng et al. [44] can predict the four priority levels with F1-measure equal to 0.63, 0.38,0.73, and 0.96 from Low to Critical levels, which means it can assign critical priority levels correctly while it faces challenges in correctly identifying Medium priority levels. This may be attributed to the proposed feature extraction method (i.e., BERT) or the KNN classifier by Meng et al. [44].

DRONE's F1-measure results show that it is struggling in assigning correct Low, Medium, and High levels to the code reviews. While it performs better with the Critical level (i.e., F1-measure equals 0.47), it remains the lowest-performing model when compared to other models in our experiment. We believe it is because of the linear regression of the Gray classifier, which is not well-suited to our dataset.

Comparing the two baselines with the result of PrAIoritize, we observe that we can improve the F1-measure for the four priority levels, with the Medium level slightly less than the rest of the priority levels. Considering the overall average F1-measure, PrAIoritize outperforms Meng et al. [44] baseline by 27.94% and DRONE baseline by approximately 194.9%. However, we only consider the textual dimension of DRONE implementation. Therefore, in cases where the dataset includes the other five dimensions (i.e., temporal, author, related report, severity, and product as proposed in DRONE paper), DRONE may surpass our model's performance.

Moreover, we studied the performance of BERT, T5, BiLSTM, and RNN in comparison to PrAIoritize performance. Among these well-known models, we notice that RNN has the poorest average F1-measure. PrAIoritize's average F1-measure outperforms BERT by approximately 4.82%, T5 by approximately 102.33%, BiLSTM by approximately 10.13%, and RNN by approxi-

Table 4.6: Comparisons of recall measures of PrAIoritize versus other classifiers and baselines

Priority Level	Low	Medium	High	Critical	Average
PrAIoritize	0.86	0.83	0.80	0.99	**0.87**
[44]	0.64	0.43	0.67	0.96	**0.68**
DRONE	0.37	0.08	0.42	0.34	**0.30**
BERT	0.85	0.81	0.73	0.97	**0.84**
T5	0.25	0.10	0.46	0.92	**0.43**
BiLSTM	0.86	0.84	0.58	0.94	**0.80**
RNN	0.72	0.93	0.37	0.95	**0.74**

Table 4.7: Comparisons of precision measures of PrAIoritize versus other classifiers and baselines

Priority Level	Low	Medium	High	Critical	Average
PrAIoritize	0.81	0.81	0.92	0.95	**0.87**
[44]	0.64	0.50	0.61	0.96	**0.68**
DRONE	0.24	0.12	0.25	0.73	**0.34**
BERT	0.67	0.79	0.95	0.99	**0.85**
T5	0.25	0.10	0.46	0.92	**0.43**
BiLSTM	0.74	0.72	0.76	0.97	**0.80**
RNN	0.81	0.56	0.84	0.96	**0.79**

mately 19.18%

Table 4.6 shows the recall measures for all baselines and models versus PrAIoritize recall measures for the four priority levels. PrAIoritize outperforms Meng et al. [44] baseline with 27.94% in terms of the recall measure and DRONE by approximately 190.0%. It also further improves the recall of BERT by approximately 3.57%, T5 by approximately 102.33%, BiLSTM by approximately 8.75%, and RNN by approximately 17.57%. Finally, Table 4.7 presents the precision measure of PrAIoritize and the rest models. Similar to F1-measure and recall, PrAIoritize significantly improves the precision of Meng et al. [44] by 27.94%, achieves a higher precision of DRONE by 155.88%, and the selected pre-trained models (i.e., BERT, T5, BiLSTM, and RNN) by 2.35%, 102.33%, 8.75%, and 10.13% respectively.

Figure 5.3 shows a detailed breakdown of the classification results produced by the PrAIoritize model, as depicted in the confusion matrix. It illustrates the

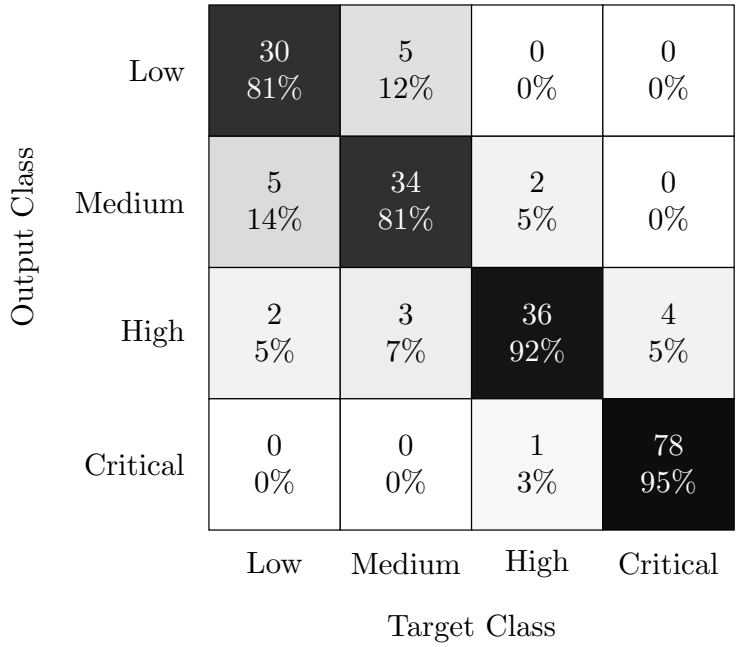


Figure 4.4: Confusion matrix for the classification results of the PrAIoritize model.

distribution of predicted labels compared to the ground truth labels. Each row in the matrix represents the actual (i.e., ground truth) class, while each column corresponds to the predicted class. For low-priority weaknesses, the model achieves a noteworthy accuracy, correctly classifying 30 instances with minimal misclassifications. Similarly, in identifying critical-priority weaknesses, the model excels, accurately classifying 78 instances with very few misclassifications. However, for medium and high-priority weaknesses, the model's performance is less consistent. While it correctly identifies a majority of medium-priority weaknesses (i.e., 34 instances), there are a notable number of misclassifications, particularly as low and high priority. Similarly, in the case of high-priority weaknesses, the model shows decent accuracy, but there are noticeable misclassifications across other priority levels. These results suggest that while the model demonstrates satisfactory performance in code weakness prioritization, there is room for improvement, particularly in reducing misclassifications across different priority

levels to enhance overall effectiveness.

Summary

- PrAIoritize effectively prioritizes code weaknesses in smart contract code reviews, outperforming both baseline models and state-of-the-art pretrained models.

4.7 Discussion and Implications

Our results demonstrate that smart contract code weaknesses can be effectively prioritized automatically during the code review process. Our proposed automated prioritization approach, PrAIoritize, leverages DistilBERT and shows superior F1-measure, precision, and recall compared to traditional models such as RNN and BiLSTM. This disparity suggests that BERT-based models possess a better understanding of the semantics embedded within code reviews, enabling them to identify nuanced weaknesses more effectively.

DistilBERT and BERT, by leveraging transformer-based architectures, excel in capturing contextual information and semantic relationships within the text. This capability allows them to grasp the intricate details and nuances present in code reviews, leading to more accurate prioritization of weaknesses. Their performance highlights the importance of utilizing advanced natural language processing (NLP) techniques for code review prioritization tasks, especially in fields where precise comprehension of textual data is paramount.

Conversely, models such as RNN and BiLSTM, although performing reasonably well, do not exhibit the same level of semantic understanding as BERT and DistilBERT. Their reliance on sequential processing may limit their ability to capture long-range dependencies and intricate semantic nuances present in code reviews. As a result, while they achieve respectable precision levels, they may not fully exploit the contextual information embedded within the code reviews, leading to slightly inferior performance compared to BERT-based models.

Additionally, automated labeling for a corpus with limited keywords is useful as a first step to reducing time and effort, allowing for the initial categorization of code reviews. However, predictive models, such as those leveraging DistilBERT, offer additional benefits by providing a more comprehensive understanding of the data and capturing nuances that automated labeling may overlook. Furthermore, predictive models can adapt to evolving datasets and provide insights that automated labeling alone may not capture, enhancing the robustness and ac-

curacy of the overall process. Furthermore, when comparing DistilBERT with T5, an intriguing observation arises. T5 differs significantly in its approach. T5 operates under the paradigm of text-to-text transfer learning, where it is trained to generate target text sequences from source text sequences in our case the code reviews. This approach allows T5 to learn complex mappings between input and output text, enabling it to effectively capture semantic relationships and contextual information within code reviews. However, compared to our approach (i.e., employing DistilBERT), T5 requires more computational resources due to its architecture, potentially leading to longer inference times. However, it is possible that increasing the number of epochs during training could improve T5's performance.

In summary, the results underscore the significance of leveraging advanced NLP techniques, particularly transformer-based models such as DistilBERT, for code weakness prioritization task in smart contracts. These models reveal superior precision by effectively understanding the semantics embedded within code reviews. While traditional models like RNN and BiLSTM perform adequately, they fall short of the nuanced understanding exhibited by transformer-based models. Additionally, T5 presents an alternative approach to capture semantic relationships within code reviews. However, its effectiveness might be influenced by its heightened resource demands and the need for extensive training.

4.7.1 Implications

Implications for Software Practitioners: For software practitioners, PrAIoritize offers a valuable approach to enhancing smart contract development and auditing processes. By leveraging PrAIoritize's capabilities, practitioners can prioritize code reviews more effectively, ensuring that critical code weaknesses are addressed promptly. The automated triage provided by PrAIoritize accelerates the identification of critical code weaknesses in smart contracts, minimizing the risk of overlooking urgent code weaknesses. Additionally, PrAIoritize facilitates early prediction of critical code weaknesses, enabling developers to promptly recognize and address potential risks. Third-party auditors can benefit from PrAIoritize's automated prioritizing capabilities, focusing their efforts on high-impact areas and providing valuable insights to project stakeholders. Furthermore, PrAIoritize improves the patch management process by automatically categorizing weaknesses based on priority, reducing development time and costs.

Implications for Researchers: Researchers can leverage PrAIoritize to advance understanding and exploration in the field of smart contract security.

By analyzing the dynamic evolution of code weaknesses and their corresponding priorities, researchers can uncover patterns that reveal changes in smart contract security landscape. PrAIoritize's predictive and prioritization capabilities can be used to gain insights into review practices and code weakness management in smart contract field. Additionally, PrAIoritize offers researchers a means to prioritize code weaknesses, facilitating subsequent studies on the correlation between developer attributes, code review methodologies, and code weakness prioritization. Future research can focus on developing multi-objective approaches to address the conflicting tasks of early code review prediction and prioritization effectively. Overall, PrAIoritize provides researchers with a powerful means to investigate code weakness prioritization in smart contracts and correlate them with socio-technical aspects of code review, thereby enhancing our understanding of smart contract security.

Implications for Tool Builders: PrAIoritize offers tool builders significant opportunities to develop advanced automation tools tailored specifically to smart contract development and auditing needs. By integrating PrAIoritize into automated bots, tool builders can create automated solutions that assist developers in predicting and prioritizing code review requests. These bots can be smoothly integrated into existing development ecosystems, such as Gerrit [13], to improve the review process by monitoring newly submitted code reviews and prioritizing them. Moreover, PrAIoritize can significantly benefit project maintainers tasked with managing large codebases featuring numerous pending code reviews, such as smart contracts and dependent Dapps. By leveraging PrAIoritize's prioritization capabilities, maintainers can efficiently manage and prioritize code reviews, even in projects with hundreds of code reviews and changes. For example, the tool can provide maintainers with a sorted list of critical weaknesses, enabling them to allocate resources efficiently for prompt mitigation. Additionally, tool builders can empower users to personalize PrAIoritize priority levels according to their review preferences, such as setting thresholds for allocated review effort or adjusting prioritization criteria. This customization ensures that PrAIoritize adapts to the specific needs of individual developers or project teams, enhancing its usability and effectiveness in diverse development environments. Overall, by leveraging PrAIoritize's capabilities, tool builders can develop innovative solutions that improve the code review process and enhance overall development efficiency in smart contract ecosystems.

These implications can significantly enhance the smart contract code review process, providing valuable assistance in efficiently identifying and prioritiz-

[13]https://www.gerritcodereview.com/

ing code weaknesses. By leveraging the capabilities of PrAIoritize, developers, and auditors can optimize their review workflows, ensuring that critical code weaknesses are addressed promptly and effectively. Furthermore, the suggested research implications offer opportunities for gaining insightful insights into how developer behavior and review practices impact code weakness management.

4.8 Related Work

This section describes related research on code review and code weakness priority prediction.

4.8.1 Prioritizing Code Reviews in Software Engineering

An early work by Greiler et al. [30] highlighted the significant challenge project maintainers face in prioritizing code review requests. This is due to the manual inspection required to determine which reviews to handle first. Several studies have attempted to address this challenge by proposing models to prioritize code reviews based on their likelihood of being addressed ([21], [33]). Another study by Zhao et al. [65] introduced a learning-to-rank approach to prioritize code changes. Moreover, Fan et al. [21] proposed a Random Forest (RF)-based approach to predict whether a code change would be addressed or abandoned. Subsequently, Islam et al. [33] enhanced the approach proposed by Fan et al. [21] with PredCR, utilizing Light Gradient Boosting Machines (LGBM) and a reduced feature set.

However, while these studies have addressed code review prioritization using multiple features, we propose prioritizing code reviews based on the semantics of the code weaknesses within the reviews and their impacts (i.e., textual features). Therefore, we employ natural language processing and large language models. Furthermore, given our focus on prioritizing smart contract code reviews with code weaknesses, we classify the reviews into four urgency-based categories for patching.

4.8.2 Code Weakness Priority Prediction in Software Engineering

Several methods have been suggested in the literature for improving the quality of software. These approaches can be grouped into duplication detection and classification, code weakness triage, and code weakness localization. Many

studies have focused on predicting the priority of code weaknesses, and some of these studies are based on deep learning. For example, Fang et al. [22] proposed a method that uses graph convolutional networks and a weighted loss function for the prediction of code weakness-patching priority. Another study by Li et al. [36] used deep multitask learning to develop an approach called PRIMA, which simultaneously learns both the code weakness category prediction task and the priority prediction task. Other studies are based on machine learning, for instance, Tian et al. [55] proposed a machine learning model for priority prediction based on features extracted from six dimensions: temporal, textual, author, related report, severity, and product. Valvida-Garcia et al. [57, 57] used the experience features of reporters to create blocking code weakness prediction models based on various classical machine learning classifiers. Zhou et al. [67] conducted a study to examine the impact of source code file feature sets on the accuracy of code weakness priority classification. The results of the study demonstrated that source code file feature sets did not perform as well as textual description features in code weakness classification. Tran et al. [56] compared different machine learning methods for evaluating the severity and priority of software code weaknesses. They suggested using an approach based on optimal decision trees to assess the severity and priority of new code weaknesses. Another study by Haung et al. [32] developed a model for predicting multi-class code weakness priority that integrates sentiment and community-oriented sociotechnical features from both users and developers. The model was validated in various scenarios, including within-project and cross-project. The results of the study indicate that including assignee and reporter features from sociotechnical perspectives can improve prediction performance.

4.8.3 Code Reviews Prioritization and Prediction in Smart Contracts

As far as we are aware, there has been no prior attempt to automatically predict the priority of code reviews and code weaknesses in smart contracts. Research into the prioritization of code reviews and weaknesses within smart contracts is still in its infancy. Some studies have attempted to define severity levels of smart contract code weaknesses (i.e., [54, 11]). However, the severity levels defined in these studies are limited to a few well-known vulnerabilities in smart contracts. The code reviews indicate that vulnerabilities in smart contracts evolve over time, just as smart contracts themselves do, leading to the emergence of new code weaknesses and resulting in new vulnerabilities. Moreover, severity levels in smart contracts are not yet standardized, similar to the priorities assigned

to code weaknesses. It is important to note that severity is determined by customers, while priority is determined by developers [55].

4.9 Threats to Validity

A potential threat to internal validity arises from considering code weakness types based only on keywords to assign priority levels, potentially resulting in inconsistencies within the same code weakness type, as priority levels may vary within it. To mitigate this, we assessed the impact of each vulnerability as listed in the code review. Additionally, an expert reviewed the assigned priorities to ensure their accuracy. An additional threat to internal validity, while we have assigned priority levels to smart contract weaknesses in our study, it is important to notice that these levels do not follow any standardized classifications due to the lack of standardization in smart contracts [52]. To mitigate this threat, our priority levels were identified based on keywords extracted from CVE records, related literature, and the reported issues on GitHub. Additionally, as the Ethereum platform and the language of smart contracts evolve, the prioritization of weaknesses may change to accommodate new advancements and features. Therefore, prioritization levels provided in this study should be considered as a starting point and may require adjustment based on future developments in the field. Additionally, future studies could enhance the methodology by incorporating severity levels (i.e., when available) into the labeling process. Another potential challenge to internal validity can be our exclusive focus on Solidity code weaknesses. This limited focus might not encompass all the code weaknesses in all smart contract languages, affecting the generalizability of our findings. It is important to be careful when applying our results to other languages, as they can differ significantly in code weaknesses and behaviors. Another potential threat to internal validity arises from the significant similarity among reported vulnerabilities in the dataset. This high level of similarity poses a unique challenge in accurately distinguishing the priority of each code weakness. To mitigate this threat, we employed techniques such as exploring the context of code weaknesses in the contract and applying DistilBERT to understand the semantics of the weaknesses. However, it is essential to interpret the results cautiously, considering the inherent complexities of the dataset.

One potential threat to external validity with our proposed approach is its generalizability. To address this, we trained and evaluated PrAIoritize on a dataset comprising two reliable sources to further verify the effectiveness of our approach and reduce the risk of threats to external validity. A potential threat to

the construct validity of our study is the choice of evaluation metrics. However, the metrics we selected (precision, recall, and F1-measure) are widely used in the literature and have been adopted in numerous previous studies, including the selected baselines [54, 44]. Another concern with the construct validity of our study is the distribution of the four levels of priority across the two data sources. While this may potentially impact the performance of PrAIoritize to some extent, the high performance across data sources with four priority levels suggests the effectiveness of our approach.

4.10 Conclusion

Smart contract vulnerabilities and weaknesses are widespread. To address these weaknesses, auditors and engineers conduct manual code reviews on platforms such as GitHub. This study aims to enhance the security and quality of smart contracts by automatically predicting the priority of smart contract code weaknesses during the code review process. We present PrAIoritize, an approach that leverages NLP and LLMs for prioritizing code weaknesses in smart contracts. Evaluation of PrAIoritize demonstrates its effectiveness in accurately prioritizing code reviews. Our approach provides a reliable means of automatically predicting code weakness priorities for smart contracts, highlighting the utility of NLP and multi-class LLMs in review prioritization. Future work will involve expanding our research by including code reviews from additional OSS projects and exploring a wider array of pre-trained models commonly used for similar tasks.

Data Availability

We made the data, scripts, and proposed model openly accessible at: `https://github.com/majdsoud/praioritize`

Acknowledgment

This work is supported by the Icelandic Research Fund (Rannís) grant number 207156-051.

Bibliography

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: a system for large-scale machine learning. In: Osdi, vol. 16, pp. 265–283. Savannah, GA, USA (2016) 166

[2] Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). In: Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6, pp. 164–186. Springer (2017) 1, 53, 89, 126, 150, 194, 196, 215

[3] Bacchelli, A., Bird, C.: Expectations, outcomes, and challenges of modern code review. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 712–721. IEEE (2013) 152

[4] Berg, J.A., Fritsch, R., Heimbach, L., Wattenhofer, R.: An empirical study of market inefficiencies in uniswap and sushiswap. arXiv preprint arXiv:2203.07774 (2022) 152

[5] Bermingham, M.L., Pong-Wong, R., Spiliopoulou, A., Hayward, C., Rudan, I., Campbell, H., Wright, A.F., Wilson, J.F., Agakov, F., Navarro, P., et al.: Application of high-dimensional feature selection: evaluation for genomic prediction in man. Scientific reports **5**(1), 1–12 (2015) 160

[6] Bilge, L., Dumitraş, T.: Before we knew it: an empirical study of zero-day attacks in the real world. In: Proceedings of the 2012 ACM conference on Computer and communications security, pp. 833–844 (2012) xiv, 153, 154

[7] Bloom, K., Argamon, S.: Unsupervised extraction of appraisal expressions. In: Advances in Artificial Intelligence: 23rd Canadian Conference on Artificial Intelligence, Canadian AI 2010, Ottawa, Canada, May 31–June 2, 2010. Proceedings 23, pp. 290–294. Springer (2010) 162

[8] Bosu, A., Iqbal, A., Shahriyar, R., Chakraborty, P.: Understanding the motivations, challenges and needs of blockchain software developers: A survey. Empirical Software Engineering **24**(4), 2636–2673 (2019) 145

[9] Bross, J., Ehrig, H.: Automatic construction of domain and aspect specific sentiment lexicons for customer review mining. In: Proceedings of the 22nd ACM international conference on Information & Knowledge Management, pp. 1077–1086 (2013) 162

[10] Chaliasos, S., Charalambous, M.A., Zhou, L., Galanopoulou, R., Gervais, A., Mitropoulos, D., Livshits, B.: Smart contract and defi security tools: Do they meet the needs of practitioners? In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, pp. 1–13 (2024) 15, 16, 145, 191, 195, 203, 205, 213, 224, 225, 226

[11] Chen, J., Xia, X., Lo, D., Grundy, J., Luo, X., Chen, T.: Defining smart contract defects on ethereum. IEEE Transactions on Software Engineering **48**(1), 327–345 (2020) 3, 13, 53, 54, 63, 66, 75, 86, 90, 92, 178

[12] Chen, J., Xia, X., Lo, D., Grundy, J., Yang, X.: Maintenance-related concerns for post-deployed ethereum smart contract development: issues, techniques, and future challenges. Empirical Software Engineering **26**(6), 117 (2021) 145

[13] Ciborowska, A., Damevski, K.: Fast changeset-based bug localization with bert. In: Proceedings of the 44th International Conference on Software Engineering, pp. 946–957 (2022) 154

[14] Clack, C.D., Bakshi, V.A., Braine, L.: Smart contract templates: foundations, design landscape and research directions. arXiv preprint arXiv:1608.00771 (2016) 1, 24, 146, 192

[15] Cosentino, V., Izquierdo, J.L.C., Cabot, J.: A systematic mapping study of software development with github. Ieee access **5**, 7173–7192 (2017) 67, 158

[16] Debole, F., Sebastiani, F.: Supervised term weighting for automated text categorization. In: Proceedings of the 2003 ACM symposium on Applied computing, pp. 784–788 (2003) 161

[17] Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018) 154, 196

[18] Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P.: Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In: Proceedings of the ACM/IEEE 42nd International conference on software engineering, pp. 530–541 (2020) 14, 67, 119, 121, 122, 134, 135, 158, 196, 226

[19] (EIPs), E.I.P.: Eip-1470: Smart contract weakness classification (swc). `https://github.com/ethereum/EIPs/issues/1469` (2024) 56, 150

[20] Ethereum: ERC721: Non-fungible token standard. `https://eips.ethereum.org/EIPS/eip-721` (2018) 148

[21] Fan, Y., Xia, X., Lo, D., Li, S.: Early prediction of merged code changes to prioritize reviewing tasks. Empirical Software Engineering **23**, 3346–3393 (2018) 15, 147, 177

[22] Fang, S., Tan, Y.s., Zhang, T., Xu, Z., Liu, H.: Effective prediction of bug-fixing priority via weighted graph convolutional networks. IEEE Transactions on Reliability **70**(2), 563–574 (2021) 15, 178

[23] Galappaththi, A., Anvik, J., Islam, R.B.: Automatically annotating sentences for task-specific bug report summarization. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1177–1179. IEEE (2021) 161

[24] Ganchev, K., Dredze, M.: Small statistical models by random feature mixing. In: Proceedings of the ACL-08: HLT Workshop on Mobile Language Processing, pp. 19–20 (2008) 164

[25] GitHub: Erc1888 array overrun . `https://github.com/energywebfoundation/origin/issues/3365` (2024) 148

[26] GitHub: erc721 contract function returning wrong value. `https://github.com/blockchain-etl/ethereum-etl/issues/319` (2024) 148

[27] GitHub: Malicious pausing the contract. `https://github.com/code-423n4/2022-09-nouns-builder-findings/issues/719` (2024) 148

[28] GitHub: Sushiswap contract. `https://github.com/sushiswap/sushiswap-interface/issues/949` (2024) 149

[29] Gousios, G., Storey, M.A., Bacchelli, A.: Work practices and challenges in pull-based development: The contributor's perspective. In: Proceedings of the 38th International Conference on Software Engineering, pp. 285–296 (2016) 147

[30] Greiler, M., Bird, C., Storey, M.A., MacLeod, L., Czerwonka, J.: Code reviewing in the trenches: Understanding challenges. Best Practices and Tool Needs (2016) 177

[31] Hu, S., Huang, T., İlhan, F., Tekin, S.F., Liu, L.: Large language model-powered smart contract vulnerability detection: New perspectives. arXiv preprint arXiv:2310.01152 (2023) 16, 146, 192, 225

[32] Huang, Z., Shao, Z., Fan, G., Yu, H., Yang, K., Zhou, Z.: Bug report priority prediction using developer-oriented socio-technical features. In: Proceedings of the 13th Asia-Pacific Symposium on Internetware, pp. 202–211 (2022) 178

[33] Islam, K., Ahmed, T., Shahriyar, R., Iqbal, A., Uddin, G.: Early prediction for merged vs abandoned code changes in modern code reviews. Information and Software Technology **142**, 106756 (2022) 15, 177

[34] Joulin, A., Grave, E., Bojanowski, P., Mikolov, T.: Bag of tricks for efficient text classification. arXiv preprint arXiv:1607.01759 (2016) 164

[35] Kim, S.M., Hovy, E.: Determining the sentiment of opinions. In: COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics, pp. 1367–1373 (2004) 161

[36] Li, Y., Che, X., Huang, Y., Wang, J., Wang, S., Wang, Y., Wang, Q.: A tale of two tasks: Automated issue priority prediction with deep multi-task learning. In: Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 1–11 (2022) 15, 178

[37] Lipton, Z.C., Berkowitz, J., Elkan, C.: A critical review of recurrent neural networks for sequence learning. arXiv preprint arXiv:1506.00019 (2015) 154

[38] Liu, B., et al.: Sentiment analysis and subjectivity. Handbook of natural language processing **2**(2010), 627–666 (2010) 161

[39] Liu, G., Guo, J.: Bidirectional lstm with attention mechanism and convolutional layer for text classification. Neurocomputing **337**, 325–338 (2019) 154

[40] Loper, E., Bird, S.: Nltk: The natural language toolkit. arXiv preprint cs/0205028 (2002) 160

[41] Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pp. 254–269 (2016) 7, 145

[42] Manning, C.D.: An introduction to information retrieval. Cambridge university press (2009) 162

[43] Mehar, M.I., Shier, C.L., Giambattista, A., Gong, E., Fletcher, G., Sanayhie, R., Kim, H.M., Laskowski, M.: Understanding a revolutionary and flawed grand experiment in blockchain: the dao attack. Journal of Cases on Information Technology (JCIT) **21**(1), 19–32 (2019) 119, 145

[44] Meng, F., Wang, X., Wang, J., Wang, P.: Automatic classification of bug reports based on multiple text information and reports' intention. In: Theoretical Aspects of Software Engineering: 16th International Symposium, TASE 2022, Cluj-Napoca, Romania, July 8–10, 2022, Proceedings, pp. 131–147. Springer (2022) 166, 170, 171, 172, 180

[45] Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781 (2013) 164

[46] National Institute of Standards and Technology: Vulnerability definition. `https://nvd.nist.gov/vuln` (2024) 3, 149

[47] Pennington, J., Socher, R., Manning, C.D.: Glove: Global vectors for word representation. In: Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), pp. 1532–1543 (2014) 164

[48] Perez, D., Livshits, B.: Smart contract vulnerabilities: Does anyone care? arXiv preprint arXiv:1902.06710 pp. 1–15 (2019) 56, 150

[49] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J.: Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of machine learning research **21**(140), 1–67 (2020) 154, 197

[50] Salton, G., Buckley, C.: Term-weighting approaches in automatic text retrieval. Information processing & management **24**(5), 513–523 (1988) 161, 162

[51] Sanh, V., Debut, L., Chaumond, J., Wolf, T.: Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. arXiv preprint arXiv:1910.01108 (2019) 146, 154, 164, 196

[52] Soud, M., Liebel, G., Hamdaqa, M.: A fly in the ointment: an empirical study on the characteristics of ethereum smart contract code weaknesses. Empirical Software Engineering **29**(1), 13 (2024) 128, 156, 161, 179

[53] Soud, M., Qasse, I., Liebel, G., Hamdaqa, M.: Automesc: Automatic framework for mining and classifying ethereum smart contract vulnerabilities and their fixes. In: 2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 410–417. IEEE (2023) 157, 198, 226

[54] Tian, Y., Lo, D., Sun, C.: Drone: Predicting priority of reported bugs by multi-factor analysis. In: 2013 IEEE International Conference on Software Maintenance, pp. 200–209. IEEE (2013) 147, 150, 166, 180

[55] Tian, Y., Lo, D., Xia, X., Sun, C.: Automated prediction of bug report priority using multi-factor analysis. Empirical Software Engineering **20**, 1354–1383 (2015) 15, 178, 179

[56] Tran, H.M., Le, S.T., Nguyen, S.V., Ho, P.T.: An analysis of software bug reports using machine learning techniques. SN Computer Science **1**(1), 1–11 (2020) 178

[57] Valdivia Garcia, H., Shihab, E.: Characterizing and predicting blocking bugs in open source projects. In: Proceedings of the 11th working conference on mining software repositories, pp. 72–81 (2014) 15, 178

[58] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Advances in neural information processing systems **30** (2017) 154

[59] Viera, A.J., Garrett, J.M., et al.: Understanding interobserver agreement: the kappa statistic. Fam med **37**(5), 360–363 (2005) 70, 169

[60] Weinberger, K., Dasgupta, A., Langford, J., Smola, A., Attenberg, J.: Feature hashing for large scale multitask learning. In: Proceedings of the 26th annual international conference on machine learning, pp. 1113–1120 (2009) 164

[61] Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014) 2, 5, 6, 7, 145, 191, 193, 194

[62] Yang, X., Lo, D., Xia, X., Zhang, Y., Sun, J.: Deep learning for just-in-time defect prediction. In: 2015 IEEE International Conference on Software Quality, Reliability and Security, pp. 17–26. IEEE (2015) 154

[63] Yang, Y., Xia, X., Lo, D., Grundy, J.: A survey on deep learning for software engineering. ACM Computing Surveys (CSUR) **54**(10s), 1–73 (2022) 154

[64] Zhang, P., Xiao, F., Luo, X.: A framework and dataset for bugs in ethereum smart contracts. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 139–150. IEEE (2020) 3, 13, 14, 15, 53, 54, 62, 63, 64, 65, 75, 90, 92, 119, 121, 122, 134, 135, 178, 199, 203, 205, 206, 213, 224

[65] Zhao, G., da Costa, D.A., Zou, Y.: Improving the pull requests review process using learning-to-rank algorithms. Empirical Software Engineering **24**, 2140–2170 (2019) 15, 177

[66] Zhao, J., Chen, X., Yang, G., Shen, Y.: Automatic smart contract comment generation via large language models and in-context learning. Information and Software Technology **168**, 107405 (2024) 146, 192

[67] Zhou, C.Y., Zeng, C., He, P.: An exploratory study of bug prioritization and severity prediction based on source code features. In: SEKE, pp. 178–183 (2022) 178

Chapter 5

Automated Detection of Logic Vulnerabilities in Ethereum Smart Contracts

Sóley: Automated Detection of Logic Vulnerabilities in Ethereum Smart Contracts Using Large Language Models

Soud, M., Nuutinen, W., and Liebel, G. 2024.

Submitted to *Journal of Systems and Software*.

Abstract

Context: Modern blockchain, such as Ethereum, supports the deployment and execution of so-called smart contracts, autonomous digital programs with significant value of cryptocurrency. Executing smart contracts requires gas costs paid by users, which define the limits of the contract's execution. Logic vulnerabilities in smart contracts can lead to excessive gas consumption, financial losses, and are often the root cause of high-impact cyberattacks.
Objective: Our objective is threefold: (i) empirically investigate logic vulnerabilities in real-world smart contracts extracted from code changes on GitHub, (ii) introduce Sóley, an automated method for detecting logic vulnerabilities in smart contracts, leveraging Large Language Models (LLMs), and (iii) examine mitigation strategies employed by smart contract developers to address these vulnerabilities in real-world scenarios.
Method: We obtained smart contracts and related code changes from GitHub. To address the first and third objectives, we qualitatively investigated available logic vulnerabilities using an open coding method. We identified these vulnerabilities and their mitigation strategies. For the second objective, we extracted various logic vulnerabilities, focusing on those containing inline assembly fragments. We then applied preprocessing techniques and trained the proposed Sóley model. We evaluated Sóley along with the performance of various LLMs and compared the results with the state-of-the-art baseline on the task of logic vulnerability detection.
Results: From our analysis of code changes on GitHub, we identified nine novel logic vulnerabilities and extended existing taxonomies with these vulnerabilities. Furthermore, we introduced several mitigation strategies extracted from observed developer modifications in real-world scenarios. Sóley outperforms existing approaches in automatically identifying logic vulnerabilities. Interestingly, the efficacy of LLMs in this task was evident with minimal feature engineering.
Conclusion: Early identification of logic vulnerabilities from code changes can provide valuable insights into their detection and mitigation. Recent advancements, such as LLMs, show promise in detecting logic vulnerabilities and contributing to smart contract security and sustainability.

5.1 Introduction

Smart contracts are self-executing programs and key components of modern blockchain such as Ethereum [6]. Smart contracts have gained widespread attention for their ability to enforce agreements without intermediaries, offering a decentralized and trustworthy approach [30]. The characteristics of smart contracts, including high financial value, immutability, and decentralized nature, have facilitated the growth of decentralized applications (DApps) that offer various functionalities in fields such as gaming, e-commerce, and more [53]. Smart contracts are typically implemented using high-level languages such as Solidity [1]. Their implementation consists of functions that define a sequence of instructions and a set of state variables [3].

Ethereum, as the first blockchain supporting smart contracts [6], has garnered significant developer interest, leading to the deployment of over 64 million smart contracts [2] and a market capitalization exceeding $300 billion [3]. On Ethereum, executing a smart contract requires fees known as *"gas"*, a unit that measures the consumption of storage and computing resources [53]. These fees, paid in Ether [4], compensate for computing resources used during contract execution [5]. Smart contracts with logic vulnerabilities are susceptible to severe consequences, including high-impact cyberattacks, execution failures due to excessive gas usage, wasted gas, and significant financial losses [7]. Identifying logic vulnerabilities is crucial for maintaining Solidity smart contracts, as emphasized by Chaliasos et al. [7]. In their study, they highlight the inefficiency of existing smart contract security tools in detecting logic-related vulnerabilities, sanity checks, and logic errors. This challenge is exacerbated by the use of inline assembly, present in approximately 23% of smart contracts [8] to add features not supported by Solidity. The complex low-level instructions of inline assembly are only partially supported by existing static analysis tools such as Slither [19] and Smartcheck [48], making these vulnerabilities difficult to detect. Logic vulnerabilities become even more critical when they exist in smart contract functions responsible for transferring Ether out of the contract, as this could potentially prevent the contract owners or users from accessing their money (i.e., Ether). A notable example is the governmental contract [2], where $2.5 million worth of Ether got locked out. Therefore, ensuring a sustainable and secure development

[1] https://docs.soliditylang.org/en/v0.8.23/
[2] https://etherscan.io/contractsVerified
[3] https://etherscan.io/
[4] Ethereum cryptocurrency
[5] https://ethereum.org/developers/docs/gas

of smart contracts is imperative, and addressing logic vulnerabilities plays a crucial role in achieving this goal. Smart contracts, with their sequential code structure and the interrelation of statements, share similarities with natural language text. Moreover, they often conform to predetermined templates and standards, leading to recognizable patterns and recurring code structures [10]. Consequently, Large Language Models (LLMs) and other Natural Language Processing (NLP) methods offer the potential for automating the detection of vulnerabilities and generating comments within smart contracts [23, 56].

Motivated by the advancements achieved by LLMs in conventional software and the inherent characteristics of smart contract code, this paper aims to answer the following research questions (RQs).

- **RQ1 - (Identification)**: To what extent do historical code changes reveal logic vulnerabilities in smart contracts?
- **RQ2 - (Detection):** How can we automatically detect logic vulnerabilities in smart contracts via LLMs?
- **RQ3 - (Mitigation):** What specific strategies do developers employ in their code changes to mitigate potential logic vulnerabilities in smart contracts?

With these RQs in mind, our methodology consists of several key steps. First, we gathered instances of smart contract vulnerabilities and their corresponding fixes from real-world Solidity code changes on GitHub. We then investigated the identified logic vulnerabilities using an open coding method [39], defining the available vulnerabilities and corresponding mitigation strategies. Subsequently, we developed Sóley to effectively detect logic vulnerabilities involving inline assembly fragments using LLMs with minimal feature engineering. Finally, we investigate the performance of various LLMs on the logic vulnerability detection task.

Our work makes the following contributions:

- **Sóley Development and Experiment.** We present Sóley, based on CodeBERTa, to detect logic-related vulnerabilities in smart contracts, particularly those involving inline assembly fragments, with minimal feature engineering. We also explore the performance of well-known LLMs on the same task for various code vulnerabilities in smart contracts.
- **Dataset.** We curated a large dataset of Solidity smart contracts labeled with vulnerability types, locations in the contracts, and other metadata.

This dataset contains 50k contracts, 428,569 instances of code vulnerabilities, and 171,180 logic-related vulnerabilities.

- **Analysis of Logic code vulnerabilities and mitigation.** We conduct an in-depth analysis of logic vulnerabilities found in code changes within smart contracts and literature, identifying nine novel vulnerabilities. Moreover, we introduce 15 mitigation strategies used by developers to address logic vulnerabilities.

We have made the models, scripts, and data utilized in this study openly available [40] to promote research replication and facilitate additional investigations by other researchers in the field.

The rest of the paper is organized as follows. In Section 5.2, we provide background information about smart contracts and the Ethereum blockchain. Our research methodology is outlined in Section 5.3. Section 5.4 presents the experiment evaluation and setup, while Section 5.5 details our findings. Section 5.6 provides discussions and implications. Section 5.7 presents the related work. Additionally, we address threats to the validity of our work in Section 5.8 before concluding in Section 5.9.

5.2 Background

5.2.1 Blockchain and Ethereum

Blockchains are transparent, decentralized ledgers of transactions [57]. A blockchain consists of a set of blocks, each containing a sequence of transactions that cannot be altered without affecting all subsequent blocks. Ethereum is a blockchain technology featuring a single virtual machine, the Ethereum Virtual Machine (EVM) [53]. Ethereum provides users with two types of accounts: a smart contract-controlled account or an Externally Owned Account (EOA) [18]. The latter allows users to initiate a transaction on the EVM to change its states or transfer cryptocurrency (i.e., Ether). Each transaction on the EVM consists of several attributes, such as a transaction hash, sender address, and receiver address. All initiated transactions are added to the blockchain after verification based on cryptographic mechanisms to the blockchain [13].

5.2.2 Smart Contracts and Solidity

Smart contracts are self-executing digital agreements, programmed with high-level languages, that require blockchain deployment in order to execute [46, 53].

They possess three key characteristics: decentralization, immutability, and the ability to handle high financial values. Being decentralized ensures that no single entity controls them, promoting transparency and trust among diverse participants [4]. A smart contract is immutable, which means that it is extremely difficult and expensive to change its code once it has been deployed on the blockchain [53]. A smart contract facilitates financial transactions, automates payment processing, enforces conditions for fund transfers, eliminates the need for intermediaries, and reduces transaction costs [13]. These features make smart contracts valuable tools for creating secure, transparent, and efficient agreements within decentralized systems.

On Ethereum, smart contracts are written in the Solidity [41] language or Vyper language [47], which includes programmable functions and state variables similar to high-level programming languages. Smart contract code is used to force strict rules, commitments, and consequences for parties under several conditions, similar to traditional legal contracts. A smart contract uses parameters within the functions to carry out transactions effectively. For instance, in a smart contract designed to manage token transfers, such as transferring tokens from one user to another, the parameters for this transaction might encompass the sender's address, the recipient's address, and the specific amount of tokens intended for transfer.

Based on the logical implementation of the smart contract, the status of variables changes during the execution of the function. In Ethereum, bytecode is generated from the Solidity contract code. The generated bytecode is then executed by the EVM [22, 50]. The generated bytecode is an assembly language that contains low-level instructions called opcodes [6]. Eventually, each deployed contract is assigned a unique address on the blockchain network. As part of their verification and agreement of the new blocks, each node participating in the blockchain executes the bytecode of the contract. Users invoke smart contracts deployed on Ethereum by sending a transaction that includes the contract's address, signature, and input parameters for the targeted function. This transaction is directed to the nearest Ethereum node for processing. Smart contracts can manifest as independent applications. For instance, smart contracts may serve as the core or the backend for decentralized applications (DApps) [27].

[6] https://ethereum.org/en/developers/docs/evm/opcodes/

5.2.3 Inline Assembly in Smart Contracts

Inline assembly is low-level code that is allowed to be used in Solidity smart contracts [41]. There are many reasons to use inline assembly in Solidity, including granting developers fine-grained control over their contracts by implementing libraries or performing operations not available in Solidity [41]. Another important reason to use inline assembly with Solidity is to optimize the contract, especially when Solidity's optimizer does not provide efficient code. Inline assembly is written in a language called Yul, which is a readable low-level language. Recently, an empirical study by Chaliasos et al. [8] showed that around 23% of 49 million contracts contain inline assembly and it is used frequently to add functionality that is not available in Solidity and for gas optimization using specific code patterns. Notably, available static analysis tools that check Solidity contracts (e.g. Slither [19], Smartcheck [48], and Osiris [49]) only partially support detecting inline assembly fragments.

5.2.4 Smart Contract Security

In spite of the benefits that smart contracts provide, they are vulnerable to various security threats [33]. In the history of smart contracts, there have been numerous cyber attacks that lead to high financial losses, with large number of smart contracts falling victim to hacking incidents leading to an estimated $6.45 billion in damages [7]. The first attack occurred in 2016 when an attack on DAO contracts led to the loss of over 3.6 million Ethers due to a reentrancy vulnerability [1]. Due to the increasing frequency of attacks, smart contract security and trustworthiness have gained significant attention from scholars, leading to numerous studies on smart contract vulnerabilities [5], smart contract analysis [48], and vulnerability detection methods [9].

Common Vulnerabilities in Smart Contracts

Among the various vulnerabilities that can compromise smart contracts [42], the following have been identified and used in the development and evaluation of Sóley:

- **Reentrancy (RE):** A well-known vulnerability in smart contract arises when a function in the contract makes an external call to another untrusted contract before resolving the previous state, which allows the untrusted contract to make recursive calls back in the original function, resulting in draining the gas from the contract [42, 48].

- **Recursive Calls in Loops (CLP):** Recursive calls within loops, particularly when combined with inline assembly, can result in excessive gas consumption and denial of service, as the contract may exceed the gas limit [4].

- **Incorrect Low-Level Calls (LLC):** low-level function calls, such as *'call'*, *'delegatecall'*, or *'staticcall'*, leading to failed executions or unintended behaviors if not checked [16].

- **Locked Ether (LE):** the contract logic only locks Ether balance all the time because of its inability to access the external library contract to transfer Ether, which increases the risk of funds becoming permanently inaccessible [42].

- **Incorrect Equality Check (IE):** Faulty equality checks in conditional statements, especially when involving inline assembly, can result in incorrect logic execution, leading to security breaches [48].

5.2.5 Large Language Models (LLMs)

Transformers enabled the development of LLMs such as BERT (Bidirectional Encoder Representations from Transformers) [14] and GPT (Generative Pre-trained Transformer) [35], which achieved remarkable performance across a wide range of NLP tasks, including text classification, sentiment analysis, language translation, and more. Based on the success of Transformers in NLP, scholars have expanded their application to programming languages. For instance, CodeBERT and CodeBERTA [21, 24] can capture semantic relationships between code and plain language. They offer generic code representations that enable various code automation tasks, providing developers with more resources for comprehending and producing code. BERT was pre-trained on a large corpus of English text, including Wikipedia. It was trained using two primary tasks: masked language modeling and next sentence prediction, where the model predicts if two given sentences follow each other in the text. Therefore, BERT understands the context of words deeply in sentences. DistilBERT is a distilled version of BERT [38]. The model's design involves the removal of token-type embeddings, contributing to a more efficient and faster processing capability. DistilBERT also simplifies the pre-training process by eliminating the Next Sentence Prediction (NSP) mechanism, while retaining the essential Masked Language Modeling (MLM) mechanism. GPT-4 [32], developed by OpenAI, was

trained on a diverse dataset that includes 8 million web pages in the English language. GPT-4 leverages advanced transformer architecture to understand and generate human-like text. T5 [36], developed by Google, can perform various natural language processing tasks, such as text translation, with high accuracy. By converting all tasks into a text-to-text format, T5 leverages a unified framework that simplifies training and improves performance across different NLP tasks. In summary, Transformer-based architectures, have enabled applications to analyze and detect code vulnerabilities using their deep contextual understanding and powerful NLP capabilities.

5.3 Method

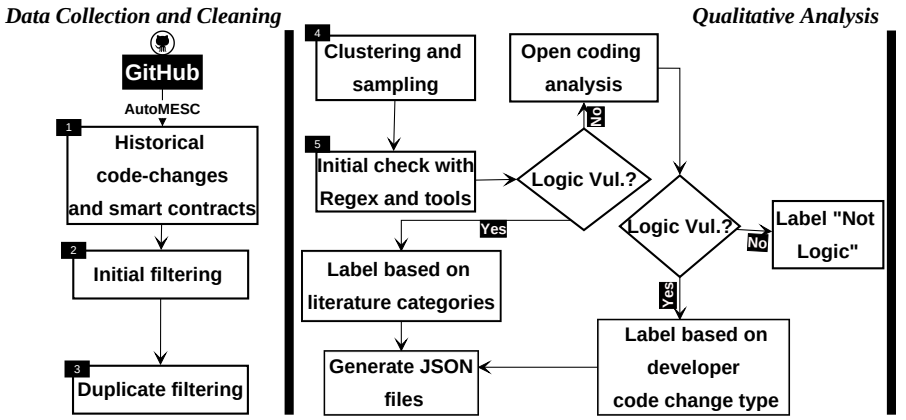


Figure 5.1: Overview of our data collection and qualitative analysis

In this section, we outline our research method. We begin by presenting the process of collecting and cleaning the smart contracts and code changes from GitHub. Afterward, we detail our qualitative approach to investigate the available logic vulnerabilities and corresponding mitigation strategies. Finally, we describe the code preprocessing, training, and evaluation phases for our proposed Sóley — automated smart contract logic vulnerability detection.

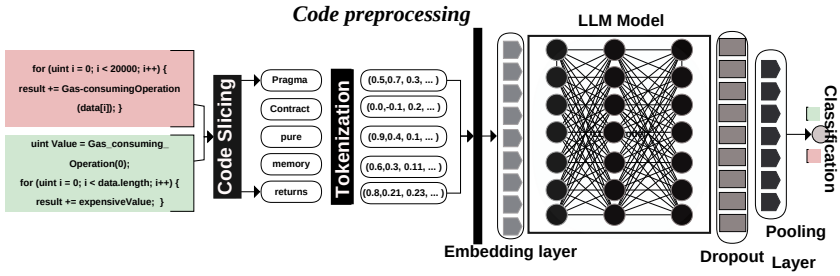


Figure 5.2: Overview of our code preprocessing and model construction

5.3.1 Data Collection

In this work, we collect an extensive dataset of real-life smart contract source code and code changes available on Open Source Software (OSS) projects hosted on GitHub using AutoMESC [43].

We selected GitHub, as it is the largest host of source code globally [51]. Moreover, smart contracts on GitHub are presented in their Solidity source code form, which includes additional essential elements such as code changes, reviews, code vulnerabilities, commits, and fixes. These components significantly enhance the comprehensibility of the code during our analysis. GitHub as a version control system that is centered around commits. A commit that fixes a code vulnerability can be viewed as a fix, comprising two crucial pairs of source code: the initial version with code vulnerabilities and the subsequent updated and, ideally, fixed version as highlighted by Zhou et al. [58]. Analyzing these commit differences (diffs) offers valuable insights into fixing the smart contract.

Figure 5.1 shows our data collection process (left side). First, we collected contracts specifically written in Solidity (identified by the .sol extension) from various projects on GitHub, including the official Ethereum repository [17] and Smartbugs [15], resulting in a corpus of Solidity smart contracts. We also extracted code changes indicating code vulnerabilities or fixed code by gathering the commit differences (diffs) for each smart contract (i.e., each .sol file), where available. However, some of these diffs were found only in comments and not in the actual contract code. We filtered out these comment-based changes, focusing solely on modifications made within the Solidity code. Our data collection and initial filtering were conducted through an automated pipeline utilizing AutoMESC [43].

5.3.2 Data Cleaning and Preprocessing

We performed data cleaning by processing the collected data from the previous step to identify potentially duplicate smart contracts and code changes. In the duplicate filtering stage, where we retained contract URLs and their source codes, we proceeded to compare Solidity contract names and URIs. Through this process, we detected possible duplicates and removed them. Then we compiled the resulting contracts to remove the non-functional contracts using the Solidity compiler [41]. Subsequently, we assigned unique hashes to distinct contracts and linked code change pairs with their corresponding contracts via AutoMESC. This approach yielded a corpus comprising a set of unique smart contracts, and metadata related to these contracts, as well as the source code of the contracts and their associated code changes when available. The resulting dataset served as the input for our qualitative analysis.

5.3.3 Qualitative Analysis Approach

Figure 5.1 shows the steps we followed to perform our qualitative analysis. Our proposed qualitative analysis approach starts with clustering the code changes based on general themes and then selecting random samples from each cluster for analysis, as we describe next.

We carefully examined a random set of 1,000 unique code changes with varying vocabulary sizes and in various contracts in the first round to understand the types of changes developers typically make to their contracts. After that, we automatically clustered the code changes based on common characteristics (i.e., frequent keywords) into eight clusters using a Python script. The resulting clusters are described in Table 5.1.

We then selected a random 10% of code changes from each cluster, except for the Upgradability and Pragma Changes clusters. We selected all 48 code changes from the Upgradability cluster considering its small size. Additionally, we selected 48 code changes from the Pragma Changes cluster, given that the initial analysis revealed no explicit logic vulnerabilities or mitigation strategies in that cluster.

After sampling and clustering steps, we started the analysis of the code changes to obtain the types of logic vulnerabilities that exist in these changes, as shown in Figure 5.1. First, we checked if the available code vulnerabilities corresponded to any of the defined logic vulnerabilities in the literature, assigning labels accordingly. We extracted the defined logic vulnerabilities from [45, 42, 54, 55] and utilized static analysis tools such as Slither [19] and

Table 5.1: Clustering of smart contract code changes

Cluster	**Total**
Standard Compliance	1099
Storage	1005
Pragma Changes	306
Contract Oracle	147
Assembly	145
Upgradability	48
Random	1805
Testing	1445

Smartcheck [48], along with regular expressions to identify logic-related vulnerabilities. The generated labels were saved in a JSON file for each contract. Two reviewers— a doctoral student expert in smart contracts and software security, and a master's degree student expert in smart contracts and machine learning— examined the regular expressions in three rounds. If no logic vulnerability was identified in the code change, we employed an open coding method for analysis. The reviewers manually inspected the code changes, related contracts for context, and any developer comments. In cases where the code change belonged to a new logic vulnerability not identified by the literature, we relied on developer comments or the type of code change to assign it a name. Otherwise, we labeled it as a non-logic-related vulnerability. The qualitative analysis resulted in labeled JSON files containing contracts, code changes, and their corresponding logic vulnerabilities, enabling us to extract categories of logic vulnerabilities and their mitigation strategies. To support our automated logic vulnerability detection, a substantial amount of data is needed. To achieve this, we utilized regular expressions on the contracts we had, in addition to employing various analysis tools mentioned earlier to identify known logic vulnerabilities in smart contracts. If the tools' output aligns with the output from the regular expressions, we automatically labeled the vulnerability and generated JSON files. Each JSON file contains information about the detected lines with code vulnerabilities, including the vulnerability type, contract filename, and metadata. In the manually labeled JSON file, we included an additional attribute that explains why the detected code change is considered a code vulnerability. This explanation facilitates replication and verification by the research community.

5.3.4 Code Preprocessing and Granularity Level Selection

Solidity contracts, despite sharing identical tokens, may vary in cleanliness or susceptibility to code vulnerabilities. This highlights the need for a detailed analysis of individual lines of code, rather than relying solely on a top-down analysis of entire contracts. Moreover, previous research indicates that processing entire files for classification has limitations in effectively identifying code vulnerabilities [28].

Therefore, our study adopts a fine granularity approach similar to that of Wartschinski et al. [51] by examining code tokens [7] responsible for logic vulnerabilities and their contextual usage by considering sequences of related code tokens. This approach enables us to precisely pinpoint the location of code vulnerabilities within sequential code, considering the interdependence of statements. Considering a specific token as a code vulnerability may result in many false positives. Instead, we assume that a token signifies a code vulnerability when utilized in a particular way. Therefore, we consider the surrounding tokens and lines, based on the lines available in the generated JSON files.

5.3.5 Code Slicing and Tokenization

After determining our study's granularity level, we proceeded with the code-slicing phase [52] as shown in Figure 5.2. This phase involved partitioning the code into two categories: negative slices containing code lines demonstrating logic vulnerabilities, and positive slices containing code lines without such vulnerabilities. We iterated through each contract and extracted the negative slices along with the corresponding type of logic vulnerability and the positive slices. The tokenizer utilized in our study is a byte-level BPE tokenizer trained on the corpus using Hugging Face tokenizers [8]. Sóley operates on the full source code of the sliced smart contract, preserving the code exactly as it is. To transform a slice of code into a numerical representation, the tokenizer first splits the code into a list of Solidity code tokens, including keywords (e.g., *"contract"*, *"function")*, identifiers (e.g., variable names), operators (e.g., *"+"*, *"-"*), and other language-specific constructs. The tokenizer provides a dedicated lexical scanner for various source codes, including Go, Python, and Java. For instance, it splits a line such as *"function transfer(address _to, uint256 _value) public returns (bool success)"* into the tokens: *"function"*, *"transfer"*, *"("*, *"address"*, *"_to"*, *","*,

[7] fundamental building blocks of the code, such as keywords, identifiers, operators, and literals.

[8] https://huggingface.co/huggingface/CodeBERTa-small-v1

"uint256", *"_value"*, *")"*, *"public"*, *"returns"*, *"("*, *"bool"*, *"success"*, *")"*, *";"*. Each of these tokens is then embedded, i.e., represented by a numeric vector. Consequently, a full section of code is transformed into a series of numbers.

5.3.6 Model Selection and Construction

Source code inherently consists of sequential data, where the effect of each statement depends on surrounding instructions. Hence, we need a model capable of learning features associated with code vulnerabilities from sequences of code tokens. Machine learning-based models, especially transformers, excel at representing the sequential nature of code while capturing its semantics. These models are adept at tasks similar to our task, as they can effectively model code and have been successfully utilized in similar contexts. Therefore, we selected CodeBERTa [24] as the model for this work. The chosen model consists of 6 layers and 84 million parameters. It is a RoBERTa-like transformer model that shares the same number of layers and heads as DistilBERT. The model was initialized with default settings and trained from scratch on the full corpus.

The selected CodeBERTa model architecture includes a series of layers and components designed to process input data effectively. The model begins with embedding layers that map input tokens to high-dimensional representations. The first embedding layer has a vocabulary size of 52,000 and outputs 768-dimensional embeddings, while the second embedding layer has a vocabulary size of 514 and outputs 768-dimensional embeddings.

Following the embedding layers, dropout layers are applied to prevent overfitting by randomly setting a fraction of input units to zero during training. Layer normalization layers are then applied to normalize the outputs of the previous layers, ensuring stable training. The model also includes multiple convolutional layers, which perform convolution operations on the input embeddings. These convolutional layers are interleaved with dropout layers and layer normalization layers to improve the model's ability to capture complex patterns in the data.

An activation function called NewGELUActivation is applied after every second convolutional layer. After the convolutional layers, additional dropout and layer normalization layers are applied to further regularize the model. Finally, it has a linear layer that maps the 768-dimensional input to a 6-dimensional output for multi-class classification.

5.3.7 Model Training

From the preprocessed data, we randomly selected 1,000 samples of five smart contract vulnerability types, all containing inline assembly fragments. These vulnerability types were chosen based on their prevalence and significance in existing literature [7, 42], to provide a diverse representation of smart contract vulnerabilities. However, for a sixth logic vulnerability type, we could only select 592 samples because these were the only ones with inline assembly while excluding those without. Inline assembly is categorized as logic-related by [54], so we included all vulnerabilities with inline assembly fragments to address the gap where existing tools fail to detect these vulnerabilities (*see* Section 5.7).

Our selection comprises diverse vulnerability types, some easily detectable while others require nuanced comprehension and deep semantic understanding, ensuring the comprehensiveness of our selection. The diversity in our selection highlights the critical importance of addressing a broad range of vulnerabilities for thorough coverage.

To ensure consistency in the training process and assess the accuracy of our model and baseline models, we maintained identical hyperparameters throughout. The length of text sequences was limited to 512 tokens, and a batch size of 10 was employed. These parameters are interrelated, as they dictate the memory requirements for each batch. While a higher batch size can expedite training, it also necessitates more memory. For optimization, we utilized the AdamW optimizer with a learning rate of 5e-5 and epsilon of 1e-8, as prescribed in the original GPT-2 paper. All models underwent 10 epochs of training, with evaluation statistics recorded at each iteration. Training larger models locally presents challenges due to their substantial memory requirements. However, we mitigated this issue by utilizing smaller models with reduced batch sizes, enabling training on laptops with 16 gigabytes of RAM and 4 gigabytes of VRAM. Although training times were longer compared to GPU clusters, attempting to initialize larger models on GPUs risked memory constraints. While decreasing sequence length and batch sizes helped address this challenge, it also resulted in longer training times. For training, we split the data into a 75/25 ratio for training and evaluation, following the default setting of the 'train_test_split' function from scikit-learn.

5.4 Experiment Evaluation

In this section, we describe our experiment and evaluation measures. Additionally, we present the selected baseline.

5.4.1 Metrics and Evaluation

We trained each of the models on the same dataset, with the same settings and hyperparameters, and then ran an evaluation benchmark from the scikit-learn [9] package. For the evaluation, we used precision, recall, and F1-measure, as well as an accuracy calculation, all from the scikit-learn package. Accuracy was determined by accumulating correct predictions throughout the evaluation process and computing the average accuracy score at its conclusion.

$$\text{Accuracy} = \frac{\text{Number of Correct Prediction}}{\text{Number of Samples}}$$

The precision value describes the model's ability to not falsely label negative samples as positive. Recall, on the other hand, measures the model's ability to accurately capture all positive samples.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

Finally, the F1-measure represents the harmonic mean of precision and recall. This metric provides a balance between precision and recall, considering both false positives and false negatives.

The accuracy in this case reflects the overall performance over all the samples, while precision, recall, and F1-measure were evaluated for each type of vulnerability separately. This approach highlights whether the model struggles in detecting specific types of logic vulnerabilities.

5.4.2 Baseline Selection

Baseline 1: We employ the base model used by Sun et al.[44] for the task of detecting six types of smart contract vulnerabilities, including reentrancy and flow vulnerabilities. The selected model leverages various implementations of

[9] `https://scikit-learn.org/stable/`

the BERT architecture. This model has been widely recognized as a ubiquitous baseline in NLP experiments for its widespread use and robust performance, as noted by Rogers et al. [37]. We adopt this baseline model for its application in detecting vulnerabilities in smart contracts. Following the description in the paper, we utilized the base BERT architecture, as the original code was not provided.

5.5 Results

This section presents our research findings and addresses the proposed research questions. We begin by analyzing logic-related vulnerabilities in real-world smart contract code changes. Next, we evaluate Sóley's automated detection capabilities across various vulnerability types and compare the results with other well-known LLMs performing the same task. Finally, we summarize the strategies developers employ to mitigate potential logic vulnerabilities.

Table 5.2: Publications discussing or identifying logic vulnerabilities in smart contracts

Papers	**Venue**	**Logic Vulnerabilities**
Sun et al. [45]	ICSE	Defined nine scenarios of known logic vulnerabilities in smart contracts.
Soud et al. [42]	EMSE	Identified four logic vulnerabilities in smart contracts.
Chaliasos et al. [7]	ICSE	Tested existing smart contract tools for logic vulnerabilities, errors, and sanity checks.
Zhang et al. [54]	ICSME	Identified eight logic vulnerabilities in smart contracts and categorized them into four categories.

5.5.1 Answers to (RQ1): Logic Vulnerability Types

To address the RQ1, regarding logic vulnerabilities in our dataset, we begin by conducting a qualitative analysis of the logic-related vulnerabilities identified by the selected tools in our dataset. Overall, the total number of vulnerabilities in our dataset is 428,568. Of these, 171,180 are logic-related code vulnerabilities.

We present a comprehensive analysis of the vulnerabilities available in our dataset in the Appendix. Before starting our manual analysis, we reviewed

Table 5.3: Identified logic vulnerabilities in the literature, along with their definitions and corresponding references

Vulnerability	**Description**	**Ref.**
Unauthorized Transfer	A smart contract allows assets to be transferred without proper authorization.	[45]
Risky First Deposit	The first deposit into a contract sets critical parameters that can be exploited by the first user.	[45]
Price Manipulation	Attackers manipulate asset prices by exploiting the liquidity pool algorithms in decentralized exchanges using Automated Market Makers.	[45]
Front Running	An attacker places a similar transaction with higher fees to front-run a pending transaction.	[45]
Wrong Interest Rate Order	Interest rates are applied in an incorrect sequence, leading to misallocation of interest payments.	[45]
Wrong Checkpoint Order	Errors in the sequence of recorded checkpoints within a contract.	[45]
Slippage	The difference between the expected price of a trade and the actual price executed.	[45]
Greedy Contract	A smart contract that consumes excessive resources (e.g., gas) during function calls.	[42]
DoS by External Contract	A smart contract relies on an external contract that can be manipulated or fails to respond.	[42]
Call to the Unknown	A smart contract calls a function on an unknown or user-specified address without proper validation.	[42], [54]
Transaction Order Dependency	The outcome of a contract depends on the order of transactions within a block.	[42], [54]
Using Assembly in the Constructor	Using low-level assembly language in a constructor to return data, bypassing typical restrictions.	[54]
Specify Function Variable as Any Type	Using a generic or ambiguous data type for function parameters.	[54]
DoS by Gas Limit	Exploiting the gas limit by creating situations where the required gas exceeds the block gas limit.	[54]
DoS by Complex Feedback Function	Invoking a function involving complex computations or recursive calls.	[54]
DoS by Non-Existent Function	Calling a function that does not exist, causing the transaction to fail and revert.	[54]
Storage Overlap Attack	Different variables or mappings share the same storage slot due to incorrect definitions.	[54]

publications on logic vulnerabilities in smart contracts to identify known vulnerabilities, as summarized in Table 5.2. We identified 17 logic vulnerabilities from the literature. We present the list of identified logic vulnerabilities along with their definitions and corresponding references in Table 5.3.

We then proceeded to manually label 614 code changes sampled from the clusters we have as mentioned in the method section 5.3, from which we were able to define nine logic vulnerabilities. In the following section, we explain each defined logic vulnerability. While certain vulnerabilities may manifest in various forms within smart contract code, due to space constraints in this paper, we present code examples when needed. Additional examples and various forms of these vulnerabilities can be found in our dataset, where the full contracts and code changes are available.

```
1
2   function setSecret(uint256 _secret) public {
3         require(isAdmin[msg.sender], "Only admin can set secret");
4
5         // Some code for setting secret here
6     }
7 function setSecretWithAssembly(uint256 _secret) public {
8     assembly {
9         // Load the isAdmin mapping slot
10        let slot := sload(isAdmin.slot)
11
12        // Logic-related vulnerability
13        //Load the value from the mapping using slot and msg.sender
14        let isAdminValue := sload(add(slot, mul(and(isZero(shr(96,
      calldataload(0))), 0xffffffff), 0x20)))  }}}
15
16
```

Listing 5.1: Bypassing Solidity access control via inline assembly

Bypassing Solidity access control via inline assembly

Inline assembly in Solidity allows developers to directly integrate assembly code within their Solidity contracts to effectively execute certain operations with less gas cost and added flexibility [8] (*see* the Background section 5.2). However, one significant risk is that inline assembly can bypass Solidity's access control mechanisms, affecting the logic and flow of the contract. Solidity's access control mechanisms typically depend on modifiers or conditional statements in Solidity code to enforce restrictions on function calls based on certain conditions [41].

Attackers or unauthorized users could potentially use inline assembly logic to gain unauthorized access to restricted function logic, even if they do not meet the specified conditions for access according to the Solidity code. We illustrate this vulnerability with an example in Listing 5.1. In the example, we have a *"setSecret"* function in Solidity that utilizes an access control mechanism, specifically employing a *"require"* statement to ensure that only admin users can set a secret. No other user should have the privilege to set the secret, making the admin the sole entity with this capability. However, we also have a *"setSecretWithAssembly"* function that bypasses the implemented access control mechanism. This is achieved by directly accessing the *"isAdmin"* mapping slot using inline assembly. Consequently, an unauthorized user can call *"setSecretWithAssembly"* and set the secret without being checked against the admin role. This leads to the unintended usage of the logic implemented in the contract and eventually compromises its security.

State manipulation via inline assembly

This vulnerability involves manipulating state variables within inline assembly, particularly under conditions that may not be immediately apparent. Such actions can lead to state inconsistencies within the logic of the contracts. These inconsistencies introduce uncertainty into the contract functionality and logic flaws. Consequently, developers and auditors may face difficulties in fully assessing the contract's logic and behavior. In Listing 5.2, the *"conditionalManipulation"* function employs inline assembly to modify the *"manipulatedValue"* only if the input _value exceeds a specific threshold, in this instance, 1000. This manipulation has the potential to disrupt the logic of the *"withdraw"* function, which verifies *"manipulatedValue"* before permitting withdrawals. If *"manipulatedValue"* is altered to an unexpected value, the *"withdraw"* function reverts, which shows how state manipulation can alter the logic of the contract.

```
1
2  function conditionalManipulation(uint256 _value) public {
3      assembly {
4          // Conditionally store value in manipulatedValue's slot
5          if gt(_value, 1000) {
6              sstore(manipulatedValue.slot, _value)}    }
7      emit Manipulation(_value);}
8
9  function withdraw(uint256 _amount) public {
10     require(balances[msg.sender] >= _amount, "Insufficient balance");
11     require(totalSupply >= _amount, "Insufficient total supply");
```

```
12
13      // Check if manipulatedValue has been set to an unexpected value
14      if (manipulatedValue > 1000) {
15          revert("State manipulated, cannot proceed");}
16      balances[msg.sender] -= _amount;
17      totalSupply -= _amount;
18      payable(msg.sender).transfer(_amount);
19      emit Withdraw(msg.sender, _amount);}
20
```

Listing 5.2: State Manipulation via Inline Assembly

Incomplete Standard function implementation

This vulnerability occurs when a smart contract fails to implement all the required functions of a standard or modifies them, resulting in an incomplete adherence to the standard [11]. As a consequence, the contract's logic is compromised, potentially leading to unexpected behavior when interacting with other contracts that assume full compliance with the standard.

Consider an ERC-20 [11] token contract that does not properly implement the "transferFrom" function, leading to a vulnerability that can cause unexpected behavior when transferring tokens between addresses. Therefore, it can potentially compromise the logic of other smart contracts or dApps built on top of the ERC-20 token standard. In this case, the logic of the ERC-20 token contract itself is compromised due to the improper implementation of the "transferFrom" function. For instance, if the "transferFrom" function fails to properly deduct the transferred token amount from the sender's balance or update the recipient's balance, it can lead to inconsistencies in token balances and potentially allow unauthorized token transfers. As a result, the contract's core logic for managing token transfers and maintaining token balances may become unreliable, impacting the overall functionality and security of the token contract.

Incorrect validation of oracle data

In blockchain, oracles serve as crucial links between off-chain data sources and on-chain smart contracts [41]. They facilitate the flow of essential data inputs to smart contracts, enabling them to execute predefined actions based on real-world events. However, when the validation process for oracle data is flawed, or when a contract overly relies on a single oracle, it introduces vulnerabilities into

the contract's logic. For instance, if a contract fails to adequately validate data obtained from an oracle, it may accept inaccurate or malicious inputs, leading to incorrect contract execution. Similarly, depending solely on one oracle introduces a single point of failure, making the contract vulnerable to manipulation. In both cases, the contract's logic becomes compromised, as it cannot ensure the integrity of the data it relies upon for decision-making. In Listing 5.3, the *"updatePrice"* function does not validate the *"newPrice"* received from the oracle. As a result, any erroneous or malicious data provided by the oracle will be accepted by the contract logic and stored without verification. This oversight can result in unexpected logic outputs or other unintended behaviors, particularly if the contract relies on the price data for critical decision-making processes.

```
interface Oracle {
    function getPrice() external view returns (uint256);}
contract PriceConsumer {
    address public oracle;
    uint256 public price;

    event PriceUpdated(uint256 newPrice);
    constructor(address _oracle) {
        oracle = _oracle;}

    function updatePrice() public {
        // Fetch price from the oracle
        uint256 newPrice = Oracle(oracle).getPrice();
        // No validation on the price data received from the oracle
        price = newPrice;
        emit PriceUpdated(newPrice); }
    function getPrice() public view returns (uint256) {
        return price;} }


```

Listing 5.3: Incorrect Validation of Oracle Data

Stale oracle data

This vulnerability arises when a smart contract relies on outdated data from oracles, leading to logical decisions based on obsolete information. For instance, trading contracts dependent on price feed oracles may execute transactions using outdated prices, resulting in unwanted logic. Moreover, inadequate handling of unexpected or malformed responses from oracles can cause unintended logic. For example, if an oracle returns a negative or zero value unexpectedly, it may

disrupt the contract's behavior. Therefore, a contract logic should gracefully handle such scenarios is crucial for maintaining logic integrity and preventing undesirable outcomes.

Enumerating storage slot logic

The logic of enumerating slots in the contract storage is significant. Developers can enumerate their contract storage slots using a pattern that is well-known and predictable. Therefore, attackers can access or guess the storage content and manipulate it in their favor. This manipulation can disrupt the intended flow of the contract logic by altering critical data stored in specific storage slots. For instance, using common offsets for storing data crucial for the contract logic enables attackers to target specific storage slots and thus execute the logic as they wish.

Insecure upgrade authorization logic

This vulnerability concerns the logic managing the authorization of contract upgrades, allowing unauthorized parties to approve or trigger upgrades. It may arise from weak or predictable authorization mechanisms, which are easily bypassed by users or unauthorized entities. Consequently, this can lead to unauthorized modifications, manipulation of contract logic, or even a complete contract takeover by malicious actors.

Inconsistent state transition checks

This logic-related vulnerability can occur during contract upgrades. It involves inadequate checks for the contract's current state or version, which can lead to unexpected behaviors in the execution of the contract logic. For instance, if the contract fails to properly verify its current state or version before initiating an upgrade, it may result in inconsistencies or vulnerabilities in the contract's logic. As a consequence, the contract may experience malfunctions, data corruption, or unexpected behaviors due to incorrect state transition checks during the upgrade process.

Insecure rollback mechanisms

This vulnerability occurs during the contract upgrading process, particularly when the implemented rollback mechanisms are inadequate or insecure. This can result in vulnerabilities or unexpected logical execution if upgrade failures or

rollbacks occur. Rollback mechanisms in smart contracts typically refer to the ability to revert changes made to the contract state in case of errors or failures. For example, an incorrect implementation of robust rollback mechanisms may fail to properly revert state changes and ensure contract logic integrity in the event of upgrade failures or rollbacks, potentially leaving the contract in an inconsistent state. The impact of this vulnerability can manifest as contract malfunction due to inadequate rollback mechanisms that fail to properly handle upgrade failures or rollbacks.

Summary

- Code changes provide valuable insights into logic vulnerabilities, as evident from identifying various logic-related vulnerabilities. However, given the intricate nature of these vulnerabilities, it is essential to analyze code changes alongside the contract's source code and related contracts to gain a comprehensive understanding.
- Manual analysis of code changes, particularly when examining the business logic, proved challenging. Nonetheless, it revealed novel logic vulnerabilities, highlighting the depth of vulnerabilities that automated methods may overlook.
- Logic vulnerabilities in smart contracts range from syntactically detectable vulnerabilities, which are easily identified by tools or using regular expressions, to semantic-specific vulnerabilities that demand a profound understanding of the semantics of the contract and its business logic and context.

5.5.2 Answers to (RQ2): Automated Detection of Logic-Related Vulnerabilities

To answer RQ2, we utilized labeled vulnerabilities as described in Section 5.3. We split the data into a 75/25 ratio for training and evaluation, following the default setting of the train_test_split function from scikit-learn[10]. The training method followed is outlined in Section 5.3. During the training process, we evaluated our approach and related models on an unseen test set to assess their accuracy in detecting various types of logic and general vulnerabilities within the contract code.

[10]`https://scikit-learn.org/stable/`

Table 5.4: Sóley performance metrics for different vulnerability categories

	Precision	Recall	F1
Reentrancy (RE)	0.89	0.88	0.89
Uninitialized Local Variables (UL)	0.93	0.93	0.93
Recursive Calls In Loops (CLP)	0.92	0.95	0.94
Incorrect Low-Level Calls (LLC)	0.94	0.93	0.94
Locked Ether (LE)	0.97	0.91	0.94
Incorrect Equality Check (IE)	0.81	0.88	0.84

To effectively detect logic vulnerabilities within smart contracts, LLMs require a substantial number of labeled instances comprising numerous labeled instances of such vulnerabilities. Instead of a binary classification approach (vulnerable or not), we need multi-class classification to pinpoint the specific type of vulnerability present, if any. We have curated a selection of vulnerabilities, all of which include inline assembly fragments and are categorized as logic-related by [54], which demand nuanced comprehension of context and semantics. These include Locked Ether (LE) and Incorrect Equality Check (IE). In contrast, we have included three vulnerabilities with inline assembly fragments (i.e., Uninitialized Local Variables (UL), Recursive Calls in Loops (CLP), and Incorrect Low-Level Calls (LLC)). In addition, we have included Reentrancy (RE) vulnerability with inline assembly, as a recent empirical analysis by Chalias et al. [7] highlighted that all preventable attacks on smart contracts were related to RE vulnerabilities. Moreover, we have omitted business logic-related vulnerabilities due to labeling challenges and the unique nature of each contract's logic. Our primary objective is to detect well-known logic vulnerabilities with inline assembly fragments that depend on contextual understanding and compare the performance of LLMs in automatically detecting them.

The results in Table 5.4 show that Sóley excels in detecting Uninitialized Local Variables (UL), Recursive Calls in Loops (CLP), and Incorrect Low-Level Calls (LLC). Precision and recall values consistently above 0.90, resulting in an F1-measure of 0.93 and above. This reveals a balanced performance, effectively identifying true positives while minimizing false positives and negatives. These vulnerabilities are characterized by recognizable patterns or behaviors. However, vulnerabilities such as LE and IE show slightly lower performance measures, indicating Sóley's challenge in accurately identifying instances of these vulnerabilities. For instance, LE vulnerabilities involve nuanced contextual considerations regarding fund locking and withdrawal conditions within smart contracts. Sim-

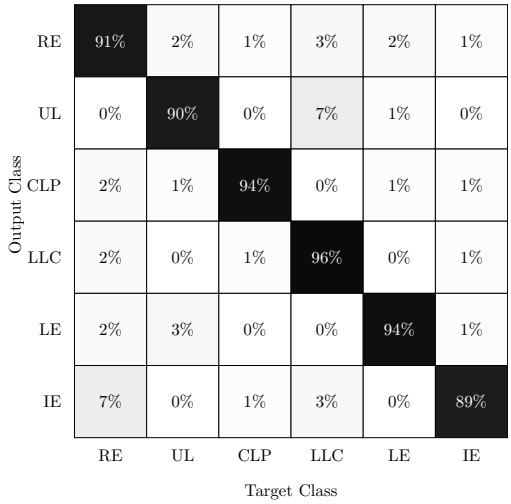


Figure 5.3: Confusion matrix for the classification results of Sóley. Note: RE refers to Reentrancy, UL to Uninitialized Local Variables, CLP to Recursive Calls in Loops, LLC to Incorrect Low-Level Calls, LE to Locked Ether, and IE to Incorrect Equality Check.

ilarly, detecting IE vulnerabilities requires a deeper understanding of how the contract handles money or value comparisons. For RE vulnerabilities, although the precision and recall are slightly lower, Sóley still achieves a high F1-measure of 0.89. We believe this is due to potential variation in the entry points based on the contract logic. Nonetheless, Sóley demonstrates strong overall effectiveness in detecting vulnerabilities within smart contracts.

In Figure 5.3, we present the confusion matrix of Sóley's classification results. We analyze the misclassification (i.e., false positives) to determine which vulnerabilities are being misclassified by Sóley. Notably, Sóley misclassifies IE as RE in 7% of cases. This suggests that many instances labeled as IE are instead identified as RE.

The resulting misclassification between these two vulnerabilities can be due to several reasons. Firstly, IE and RE vulnerabilities might share similar patterns, making it difficult for the model to distinguish between them. For example, both vulnerabilities can involve complex control or data flow, which may confuse the model. Secondly, the training data might not adequately represent

Table 5.5: Model evaluation in terms of F1-measure and accuracy (Acc.). Note: DBERT refers to DistilBERT. RE stands for Reentrancy, UL for Uninitialized Local Variables, CLP for Recursive Calls in Loops, LLC for Incorrect Low-Level Calls, LE for Locked Ether, and IE for Incorrect Equality Check.

	Epoch 10						
	RE	**UL**	**CLP**	**LLC**	**LE**	**IE**	**Acc.**
Baseline	0.78	0.84	0.87	0.91	0.89	0.66	0.84
GPT2	0.85	0.88	0.88	0.93	0.92	0.81	0.88
T5-Base	0.85	0.86	0.88	0.94	0.89	0.78	0.87
Sóley	**0.9**	**0.91**	**0.93**	**0.95**	**0.95**	**0.9**	**0.93**
DBERT	0.83	0.88	0.86	0.93	0.9	0.8	0.87

the differences between these vulnerabilities, causing the model to generalize poorly to new instances. Lastly, the simplicity of Sóley model architecture may prevent it from capturing nuanced differences between these two vulnerability types. However, since the percentage of misclassification is relatively small, we believe that there are patterns in these two vulnerabilities that share similarities in terms of data flow and the training data does not provide the model with clearly distinct features.

Additionally, we observe a 3% misclassification rate from low-level calls to uninitialized local variables. The misclassification between these two types can be explained by instances where low-level function calls may share similarities with scenarios where variables are used without being properly initialized, such as both involving memory address access, leading to confusion.

To determine how well Sóley performs in comparison to the baseline and other related models, we fine-tuned the selected models for 10 epochs to maintain fair comparison. We also used the same hyperparameters for all models and the same training evaluation percentages. Table 5.5 compares the selected models in terms of F1-measure and accuracy at epoch number 10. Our results indicate that Sóley outperforms the selected models and baseline in terms of accuracy, with a percentage of 5%-9% of improvement. We believe that is because our selected model was pre-trained on different programming languages, including object-oriented languages, that share similarities to Solidity syntax [4].

Regarding the F1-measure, we observe that all models struggle with the RE, it is a result of determining the entry point in the logic of the contracts. So mostly the model identifies functions with the contract as susceptible to RE

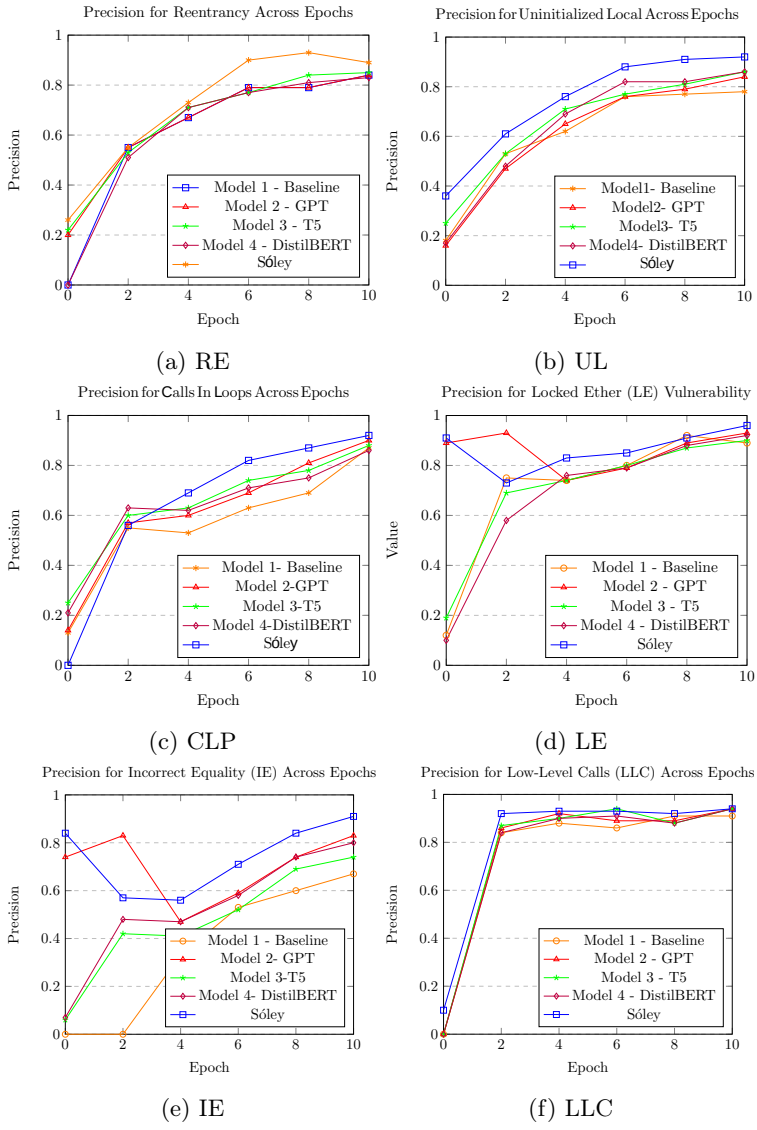
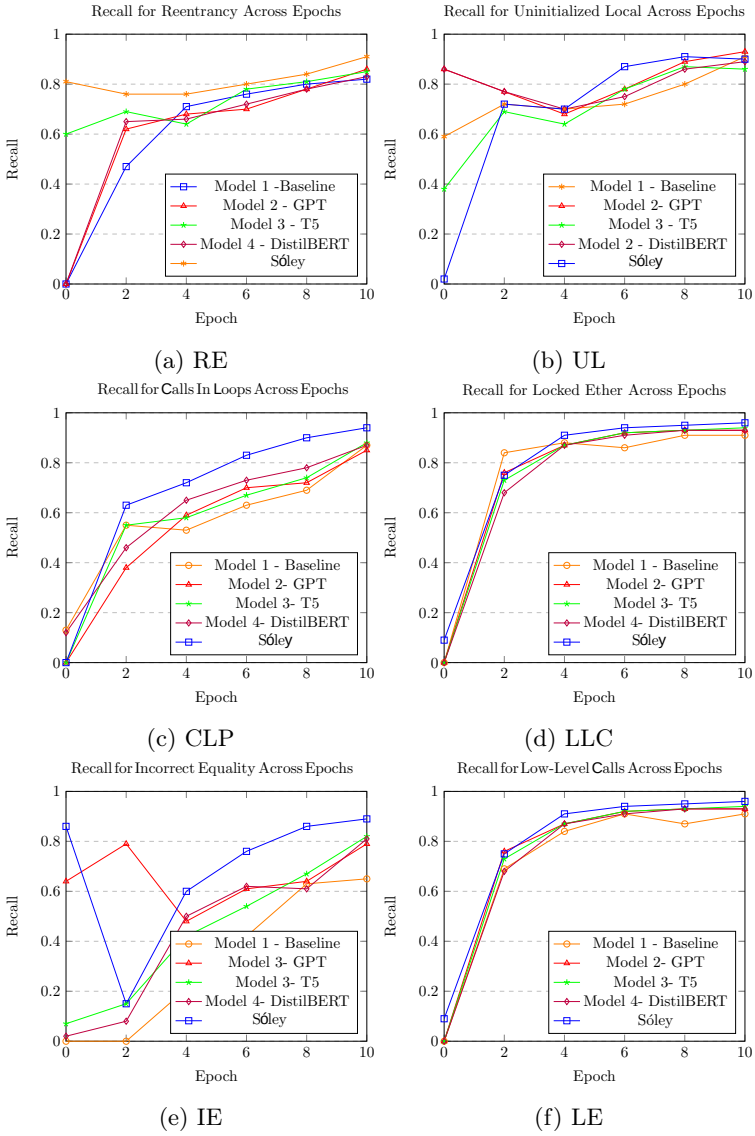


Figure 5.4: Precision rates across epochs for selected vulnerabilities.



(a) RE (b) UL

(c) CLP (d) LLC

(e) IE (f) LE

Figure 5.5: Recall rates across epochs for selected vulnerabilities.

when they are not. Most of the models also face false positives in the IE, and that is because of erroneous identification of valid equality comparisons as vulnerabilities. The low-level calls vulnerability has the highest F1 measures across all models. This vulnerability arises when low-level function calls are made without proper validation, leading to security risks such as buffer overflows [42] or injection attacks. It seems that all models recognize LLC vulnerabilities, which are the easiest to detect. Finally, we observe that all models share a close trend regarding the detection performance for all vulnerabilities at epoch 10 in terms of F1-measures.

To study the performance of the selected models across different epochs we tracked the precision for all models per vulnerability, as shown in Figure 5.4. The first six charts provide insights into the precision of various models across epochs for detecting different vulnerabilities. Across all vulnerabilities, we observe an overall increasing trend in precision as the training epochs progress. Therefore, while there may be variability in performance across different vulnerabilities, the models consistently improve in their ability to detect vulnerabilities over time, with some vulnerabilities being easier to detect than others. Similarly, recall rates in Figure 5.5 reveal a notable increase in the performance of the selected models. However, for the IE vulnerability, we see an unusual pattern where Sóley's recall starts high at 0.86, drops significantly at epoch 2, and then gradually recovers, indicating potential instability or sensitivity to early training stages. In contrast, DistilBERT exhibits a much lower starting recall for IE but demonstrates a steady improvement, highlighting its robustness in learning over time despite initial setbacks. T5 also shows a unique pattern with a slow start in recall for incorrect equality but catches up significantly by epoch 10. However, with our dataset, the models began to overfit after 10 epochs, leading us to stop at this point. We believe there is potential for further accuracy improvements with extended training and larger datasets, especially for large models such as T5 and GPT.

Summary

- Our Sóley automated detection for logic-related vulnerabilities outperforms the baseline and the state-of-the-art related LLMs in terms of accuracy, with a percentage of 5%-9% of improvement.
- Our experiments demonstrate that LLMs can effectively detect several types of vulnerabilities with minimal feature engineering. However, all models struggle with detecting Reentrancy (RE) vulnerabilities, most likely due to challenges in recognizing the contract's entry point or the insufficient number of contextually rich examples in the training set.

5.5.3 Answers to (RQ3): Mitigation Strategies in Smart Contracts

To answer this RQ, we comprehensively summarize common strategies followed by developers to address potential logic vulnerabilities based on the collected code changes available in our dataset. In the following, we provide detailed descriptions of each mitigation strategy. Moreover, many examples can be found in our dataset with the labeled code changes.

S1: Optimizing and Simplifying Bitwise Operations in Inline Assembly. This strategy involves reviewing and replacing unnecessary bitwise shift operations within inline assembly with built-in Solidity operators, such as »for bitwise right shifts. Developers simplify bitwise manipulation techniques in inline assembly and remove complex ones to reduce the potential for logic vulnerabilities arising from misunderstandings in the assembly code.

S2: Adding Checks for Low-Level Calls in Inline Assembly. Developers utilize additional checks and assertions when making low-level calls in inline assembly, as these calls can bypass the logic in Solidity and introduce vulnerabilities via unauthorized or unintended behavior.

S3:Removing Unnecessary Inline Assembly. When unnecessary inline assembly is used in the contract, we have noticed that developers either delete it or minimize the logic in the inline assembly.

S4: Invariant Validations in Upgradable Contracts. When performing contract upgrades, essential contract properties (i.e., invariants) must be preserved. These invariants include properties related to the contract such as interactions with external contracts, the contract state, and so on. Any in-

consistency in these properties may affect the dependent logic and significantly impact the whole contract, resulting in vulnerabilities and unwanted behavior. This strategy includes using assert statements to ensure that these invariants are the same across all versions and that the functionality is consistent and unchanged.

S5: Adding Validation to Ensure Withdrawals. This strategy involves developers adding validation checks to withdrawal functions within the contract logic. These checks verify if the withdrawal amount exceeds the balance of the recipient. If the withdrawal amount exceeds the balance, the transaction is reverted, and an appropriate error message is provided. Moreover, these checks are added to ensure that only authorized users can withdraw funds and that the withdrawal process is executed securely.

S6: Adding Checks for Resetting Token Allowances. This strategy is employed by developers to validate token allowance management functions to prevent unintended changes to token allowances, reducing the risk of potential logic vulnerabilities such as unauthorized token transfers. By adding specific assertions, developers ensure that resetting token allowances aligns with the contract's intended functionality.

S7: Adding Checks for Incorrect Fee Calculations. Developers modify the code to ensure accurate and reliable fee calculations by adding validation checks to fee calculation functions. This also includes adjusting the length and prefix calculations to ensure they are based on correct input values.

S8: Adding Checks for Handling Zero Balances.

Developers add checks to functions that handle token balances to ensure that zero balances are handled appropriately. These checks are added to manage cases where the balance of the recipient or the sender (*'msg.sender'*) is zero. Depending on the business logic of the contract, developers handle these cases differently, reducing the risk of potential vulnerabilities such as balance manipulation.

S9: Optimizing and Refactoring Interest Rate Calculation. Refactoring the interest rate calculation involves avoiding unnecessary intermediate steps and redundant operations. For instance, combining the calculation of interest rates for borrowing and lending into a single calculation while also adding proper checks for current interest rates and calculations.

S10: Ensuring Consistency Between Function Modifiers and State Mutability Specifiers. Addressing these types of logic vulnerabilities involves ensuring consistency between function modifiers and state mutability specifiers. This includes selecting the appropriate state mutability specifier (pure, view, payable, or nonpayable) based on the function's behavior and ensuring that

the function modifier accurately reflects the function's interaction with external contracts or accounts.

S11: Validating Selection of Operations Based on Opcode Values. When a developer decides to use operations in their logic using opcodes, they validate the selection of operations to ensure that opcode values fall within the expected range. This ensures that only authorized operations are executed, reducing the risk of potential vulnerabilities.

S12: Accurate Parameter Handling in Function Invocations. We have noticed that some functionalities within the logic of the contract require certain inputs, either from inside or externally. Developers add validations for the parameters of such functions to ensure they adhere to expected formats, ranges, and constraints. Invalid or malicious inputs can lead to unexpected logic or vulnerabilities.

S14: Minimizing and Validation of External Function Calls. External calls are checked by developers in two steps: first, by checking the return value to verify whether the operation succeeded or failed, and second, by using *'require'* statements or conditional checks to validate return values. For example, if calling an external contract to transfer tokens, a developer checks if the transfer was successful before proceeding with further operations.

S15: Removing or Limiting Complex Data Structures. Complex data structures, such as nested arrays, increase the likelihood of unintended behaviors in the contract logic. Developers remove or limit the complexity of these data structures. This mitigation strategy often entails refactoring the code to replace complex data structures with simpler alternatives, such as using flat arrays instead of nested structures. Developers also impose constraints on the depth or complexity of data structures to prevent them from becoming overly intricate.

5.6 Discussion and Implications

This section discusses the limitations and implications of our findings.

Our method has several limitations, primarily centered around the reliance on labeling through regular expressions, state-of-the-art tools, and manual labeling. Our approach includes multiple steps to identify vulnerabilities in both smart contracts and code changes. Initially, we utilize regular expressions, but this method often results in high false positives. To mitigate this, we focus on vulnerabilities that are also identified by state-of-the-art tools. However, not all vulnerabilities, especially those related to logic, can be effectively identi-

fied using regular expressions alone. Variants of vulnerability types that involve complex logic may remain undetected by this method. Furthermore, while state-of-the-art tools are robust, they may not encompass every type of vulnerability, particularly those originating from logical errors. Hence, our method includes labeling vulnerabilities based on the intersection of findings from both regular expressions and state-of-the-art tools.

In essence, there may exist vulnerabilities that neither method detects or new types of vulnerabilities that were previously unknown. Our manual checks have identified and defined new vulnerabilities. However, manually labeling a dataset as extensive as ours poses significant challenges, compounded by the absence of an automated method for double-checking identified vulnerabilities. Therefore, adopting a combined approach that integrates tools and regular expressions is necessary to effectively identify known vulnerabilities. To minimize this problem during testing and training the model, we have focused primarily on well-known logic vulnerabilities such as locked ether and incorrect equality sanity checks.

The second limitation is capturing the context of vulnerabilities. The diff and selected lines describing vulnerabilities and their context in our work may have some limitations. There are instances where vulnerabilities depend on interactions across extensive portions of code, spanning large contracts or multiple libraries included in those contracts. Our model may not fully understand the implications of such widespread dependencies, as it was not trained on these specific scenarios.

Furthermore, while we considered various types of vulnerabilities, the context in which they occur may not always be fully represented. Therefore, we focus on typical and crucial logic-related vulnerabilities defined in the literature for which we have confidence in our model's training on a substantial dataset. Moreover, our dataset is restricted to mitigations that developers have implemented in their code changes (diffs). This means that any unrecognized mitigations or ignored vulnerabilities remain unseen, creating gaps in our understanding of new vulnerabilities or mitigations. Therefore, we deliberately focused only on the mitigations and vulnerabilities that developers have fixed, similar to the approach used by Liu et al. [26]. This method helps us avoid false positives by considering only the issues that developers have addressed, but it also means we might miss vulnerabilities that developers did not recognize, fix, or mitigate.

Finally, our model exhibits faster training times due to its architecture being a simplified version of the BERT architecture. However, it is important to note that its smaller size could make it more susceptible to overfitting compared to larger models. We anticipate that a larger dataset and more extensive training

could enable larger models to achieve similar levels of accuracy.

5.6.1 Implications

For Researchers. Researchers can utilize Sóley to expand the scope of vulnerability detection in smart contracts. Utilizing minimal feature engineering techniques such as tokenization and slicing, our results show that Sóley and LLMs offer promising avenues for accurately detecting diverse vulnerabilities. We believe additional exploring and the integration of control flow or data flows with LLMs could enhance vulnerability detection by integrating more advanced feature engineering. Adding more context, such as comments alongside code snippets or additional lines, could improve the semantic understanding of LLMs and enhance their detection capabilities. Moreover, Sóley can be used to curate datasets and generate labeled datasets with more accurate and semantically rich vulnerabilities. Future research could focus on developing automated approaches that utilize code changes to identify vulnerabilities in smart contracts at an early stage. Lastly, there is a need for further exploration into automating the patching of smart contract vulnerabilities using code changes, as well as investigating the efficacy of different fix strategies commonly employed by developers.

For Practitioners and Tool Builders. Developing automated analysis tools that integrate Sóley with version control systems such as Git can improve detecting logic vulnerabilities in smart contracts. Additionally, tool builders can extend existing analysis tools by integrating Sóley to detect logic-related vulnerabilities through plugins or extensions. Furthermore, tool builders can explore ways to augment their tools' capabilities to automate the patching of identified vulnerabilities. By providing developers with code suggestions for fixing their smart contracts, which may contribute to more robust and accurate smart contracts. Leveraging LLMs and Sóley, practitioners can effectively assess, analyze, and secure smart contracts against potential vulnerabilities.

5.7 Related Work

This section describes studies related to our work, including empirical studies on logic code vulnerabilities in smart contracts, LLMs in smart contract vulnerability detection, and datasets related to our research.

5.7.1 Empirical Studies on Logic Vulnerabilities in Smart Contracts

In a study by Chaliasos et al. [7], an empirical evaluation of existing tools for smart contract security was conducted. Their evaluation indicates that these tools often generate numerous insignificant reports, leading to an overwhelming number of false positives. Moreover, Chaliasos et al. highlight the inefficiency of these tools in detecting logic-related vulnerabilities, sanity checks, and logic errors. The study also shows that practitioners identify logic-related vulnerabilities and protocol layer vulnerabilities as significant threats that are not adequately addressed by existing security tools. Soud et al. [42] defined logic vulnerabilities as inconsistencies with the contract and the programmer's intention. Among the identified logic-related vulnerabilities are Greedy Contract (i.e., locked ether), which arises when the contract logic only locks Ether; Transaction Order Dependency, occurring when a contract's logic depends on the order of transaction execution within a block; Call to the Unknown vulnerability, where a function unexpectedly invokes the recipient's fallback function, potentially introducing malicious code; and the DoS by External Contract vulnerability. Zhang et al. [55] categorize 26 smart contract vulnerabilities into three groups. The first group includes hard-to-exploit or doubtful vulnerabilities. The second group is detectable by static analysis tools. The third group involves business logic vulnerabilities, such as price manipulation, which require high-level semantical oracles and are generally undetectable by current static analysis tools. Finally, Zhang et al [54] defined logic-related vulnerabilities as flaws in the decision logic, branching, sequencing, or a computational algorithm, as found in natural language specifications or implementation language. The study identified four primary logic-related categories in smart contracts; including assembly code (i.e., inline assembly), DoS, fairness, and storage vulnerabilities.

5.7.2 Detecting Smart Contract Code Vulnerabilities

Detecting smart contract vulnerabilities has attracted significant attention recently. Several tools have emerged to analyze smart contract code. These tools can generally be categorized into two main groups: traditional static and dynamic analysis tools and tools based on machine learning and large language models (LLMs).

Solidity vulnerability detection via traditional code analysis tools. Static analysis tools such as Slither [19] and smartcheck [48] are designed to stat-

ically analyze smart contracts, detecting security vulnerabilities and bad coding patterns. Solhint [34] detects syntax-related vulnerabilities through static checks using a wide range of rules. Symbolic execution tools such as Mythril [29] identify various vulnerabilities, including overflow/underflow and tx.origin issues. Additionally, Osiris [49] facilitates integer vulnerability detection in Solidity by employing symbolic execution combined with taint analysis. Dynamic analysis tools such as Maian [31] categorize vulnerable smart contracts into three main types: suicidal contracts, prodigal contracts, and greedy contracts. Maian analyzes Solidity contracts through dynamic analysis on a private blockchain to reduce the number of false positives However, despite the effectiveness of these tools, Chaliasos et al. [7]'s empirical evaluation of these tools highlights that they do not adequately address logic-related vulnerabilities, which are often the root cause of high-impact attacks. Additionally, Zhang et al. [55] noted that more than 80% of exploitable bugs remain undetectable by automated means. Moreover, the complex low-level instructions of inline assembly are only partially supported by existing static analysis tools [7], and no existing tool in the available body of knowledge is designed to detect vulnerabilities with inline assembly fragments in smart contracts by the time of writing this paper.

Solidity Vulnerability Detection using LLMs.

Sun et al. [45] introduced GPTScan, a tool that combines GPT with static analysis. GPTScan prompts GPT to automatically recognize scenarios related to logic vulnerabilities in smart contracts. Additionally, GPT is trained to identify key variables and statements, which are then verified through static confirmation. Hu et al. [23] proposed GPTLENS based on GPT-4 as an auditor, analyzing smart contracts, and a critic, reviewing the audits. Their empirical findings indicate that GPTLENS brings significant enhancements compared to the traditional detection approaches. David et al. [12] examined the feasibility of utilizing LLMs for smart contract security audits, evaluating them on 52 compromised Decentralized Finance (DeFi) smart contracts. Their findings indicate that GPT-4 and Claude models correctly identify vulnerabilities in 40% of cases but exhibit a notable false positive rate, necessitating manual auditor involvement. Sun et al. [44] introduced ASSBert framework, designed for smart contract vulnerability classification based on active and semi-supervised bidirectional encoder representations from transformers (BERT) network. Finally, Jeon [25] proposed SmartConDetect, a static analysis tool designed to detect security vulnerabilities in Solidity-based smart contracts. Utilizing a pre-trained BERT model, SmartConDetect extracts code fragments from smart contracts and identifies vulnerable code patterns.

Existing approaches mainly detect Solidity vulnerabilities, yet few are de-

signed specifically to detect vulnerabilities within the logic and inline assembly of the contract. Although the nature of logic-related vulnerabilities, sanity checks, and inline assembly vulnerabilities, often making it more challenging to detect [7, 8].

5.7.3 Solidity Vulnerability Datasets

Several studies in the literature focused on datasets for Solidity vulnerabilities. Durieux et al. [16] curated a dataset with 69 annotated vulnerable Solidity contracts obtained from Etherscan. Their dataset includes labels specifying the location and category of the vulnerabilities. Moreover, The authors provided an unlabeled dataset containing 47,518 contracts. Soud et al. [43] constructed a dataset comprising approximately 6,500 vulnerabilities and corresponding fix pairs. Chalias et al. [8] constructed a dataset of Solidity smart contracts containing 12.4 million contracts. However, there are no available datasets specifically about logic vulnerabilities or logic-related inline assembly, nor a dataset with labeled vulnerabilities with code snippets of the vulnerable section.

5.8 Threats to Validity

This section addresses challenges to the internal, construct, and external validity. To identify validity threats in our study, we utilized the standard methodology proposed by Feldt et al. [20].

Internal Validity: Labeling of vulnerabilities in large quantities poses a threat to internal validity due to the time and resource-intensive nature of the task. To mitigate the risk of incorrectly labeling contracts, we employed regular expressions and validated the results using two established tools, SmartCheck and Slither. Another internal validity concern relates to the availability of training data. Given the specificity of our task, obtaining a large amount of suitable data related to logic vulnerabilities with inline assembly was challenging. Addressing this, we selected vulnerabilities classified in the literature as logic vulnerabilities and compared them with other types of vulnerabilities, all with inline assembly fragments. The selection of models also presents a potential threat to internal validity. To mitigate this, we chose a range of well-reviewed LLMs. Additionally, there is a concern about the reliability of the quantitative analysis, considering the potential presence of unidentified vulnerabilities within the contracts. To address this, two experts reviewed the identified vulnerabilities from the manual labeling process. Furthermore, we have made the raw data

openly accessible, allowing other researchers and users to validate the results.
Construct Validity: This could be the choice of evaluation metrics in our study. Nevertheless, we minimized this threat by selecting widely recognized metrics such as precision, recall, and F1-measure. These metrics have been extensively employed in previous research, including the baseline. Another consideration regarding the construct validity of our study is the distribution of selected vulnerabilities, which may vary in quantity within the dataset. To ensure fair training and balanced data representation, we standardized the number of instances across all vulnerabilities, despite their varying occurrences in the dataset. The consistent high performance across all vulnerabilities suggests the efficacy of our approach. Additionally, to address concerns about training fairness across models, we maintained consistency by using identical hyperparameters, training datasets, optimizers, and the same machine for all models.
External Validity: A potential challenge to external validity is associated with the concern that the Ethereum smart contracts included in this study may not be a precise representation of all smart contracts across various blockchain platforms, such as Hyperledger Fabric or other contracts on different blockchains. Consequently, our quantitative analysis and approach may not accurately reflect the code changes or fixes in contracts beyond Ethereum smart contracts.

5.9 Conclusion

Securing smart contracts has attracted many researchers and practitioners. One important aspect of improving smart contract security is addressing logic-related code vulnerabilities, which often serve as the root cause of high-impact cyber-attacks. In this work, we investigate logic-related vulnerabilities from code changes in smart contracts collected from GitHub. We introduce Sóley to detect these code vulnerabilities and compare its performance with other types of code vulnerabilities. Additionally, we explore and identify mitigation strategies for these code vulnerabilities based on developers' code changes. Sóley outperforms baselines and several LLMs in automatically identifying logic vulnerabilities. Interestingly, without requiring extensive feature engineering, Large Language Models (LLMs) performed effectively in this task, as evident from the results. In the future, we plan to explore the detection and fixing of other types of logic-related vulnerabilities.

Data Availability

Data is available at [40].

Bibliography

[1] Deconstructing thedao attack: A brief code tour (2016). URL https://web.archive.org/web/20180128074919/http:/vessenes.com/deconstructing-thedao-attack-a-brief-code-tour/ 195

[2] Governmental contract. https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck (2016) 191

[3] Antonopoulos, A.M., Wood, G.: Mastering ethereum: building smart contracts and dapps. O'reilly Media (2018) 191

[4] Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). In: Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6, pp. 164–186. Springer (2017) 1, 53, 89, 126, 150, 194, 196, 215

[5] Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). pp. 164–186 (2017). doi:10.1007/978-3-662-54455-6_8 195

[6] Buterin, V., et al.: A next-generation smart contract and decentralized application platform. white paper **3**(37), 2–1 (2014) 1, 119, 191

[7] Chaliasos, S., Charalambous, M.A., Zhou, L., Galanopoulou, R., Gervais, A., Mitropoulos, D., Livshits, B.: Smart contract and defi security tools: Do they meet the needs of practitioners? In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, pp. 1–13 (2024) 15, 16, 145, 191, 195, 203, 205, 213, 224, 225, 226

[8] Chaliasos, S., Gervais, A., Livshits, B.: A study of inline assembly in solidity smart contracts. Proceedings of the ACM on Programming Languages **6**(OOPSLA2), 1123–1149 (2022) 191, 195, 207, 226

[9] Chen, T., Feng, Y., Li, Z., Zhou, H., Luo, X., Li, X., Xiao, X., Chen, J., Zhang, X.: Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts. IEEE Transactions on Emerging Topics in Computing **9**(3), 1433–1448 (2020) 195

[10] Clack, C.D., Bakshi, V.A., Braine, L.: Smart contract templates: foundations, design landscape and research directions. arXiv preprint arXiv:1608.00771 (2016) 1, 24, 146, 192

[11] Community, D.: Erc-20, erc-721, and other smart contract standards explained (2024). URL `https://dev.to/ethereum/erc-20-erc-721-and-other-smart-contract-standards-explained` 209

[12] David, I., Zhou, L., Qin, K., Song, D., Cavallaro, L., Gervais, A.: Do you still need a manual smart contract audit? arXiv preprint arXiv:2306.12338 (2023) 16, 225

[13] Destefanis, G., Marchesi, M., Ortu, M., Tonelli, R., Bracciali, A., Hierons, R.: Smart contracts vulnerabilities: a call for blockchain software engineering? In: 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), pp. 19–25. IEEE (2018) 194

[14] Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018) 154, 196

[15] Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P.: Smartbugs. `https://github.com/smartbugs/smartbugs` 198

[16] Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P.: Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In: Proceedings of the ACM/IEEE 42nd International conference on software engineering, pp. 530–541 (2020) 14, 67, 119, 121, 122, 134, 135, 158, 196, 226

[17] Ethereum: Ethereum GitHub Repository (2024). URL `https://github.com/ethereum`. Accessed: 06.2024 198

[18] Ethereum Docs: Ethereum accounts (2023). URL `https://ethereum.org/en/developers/docs/accounts/` 5, 193

[19] Feist, J., Grieco, G., Groce, A.: Slither: a static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), pp. 8–15. IEEE (2019) 15, 128, 191, 195, 199, 224

[20] Feldt, R., Magazinius, A.: Validity threats in empirical software engineering research-an initial survey. In: Seke, pp. 374–379 (2010) 37, 226

[21] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al.: Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020) 196

[22] Hirai, Y.: Defining the ethereum virtual machine for interactive theorem provers. In: Financial Cryptography and Data Security: FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers 21, pp. 520–535. Springer (2017) 194

[23] Hu, S., Huang, T., İlhan, F., Tekin, S.F., Liu, L.: Large language model-powered smart contract vulnerability detection: New perspectives. arXiv preprint arXiv:2310.01152 (2023) 16, 146, 192, 225

[24] Hugging Face: CodeBERTa-small-v1. `https://huggingface.co/huggingface/CodeBERTa-small-v1`. Accessed on: 2024-05-21 196, 202

[25] Jeon, S., Lee, G., Kim, H., Woo, S.S.: Smartcondetect: Highly accurate smart contract code vulnerability detection mechanism using bert. In: KDD Workshop on Programming Language Processing (2021) 16, 225

[26] Liu, K., Kim, D., Bissyandé, T.F., Yoo, S., Le Traon, Y.: Mining fix patterns for findbugs violations. IEEE Transactions on Software Engineering **47**(1), 165–188 (2018) 222

[27] Metcalfe, W., et al.: Ethereum, smart contracts, dapps. Blockchain and Crypt Currency **77**, 77–93 (2020) 194

[28] Morrison, P., Herzig, K., Murphy, B., Williams, L.: Challenges with applying vulnerability prediction models. In: Proceedings of the 2015 Symposium and Bootcamp on the Science of Security, pp. 1–9 (2015) 122, 201

[29] Mueller, B.: Smashing ethereum smart contracts for fun and real profit. In: 9th Annual HITB Security Conference (HITBSecConf) (2018) 15, 128, 225

[30] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Decentralized business review (2008) 1, 191

[31] Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th annual computer security applications conference, pp. 653–663 (2018) 15, 128, 225

[32] OpenAI: Gpt-4: Technical report (2023). URL `https://openai.com/research/gpt-4`. Accessed: 2024-06-07 196

[33] Perez, A.J., Zeadally, S.: Secure and privacy-preserving crowdsensing using smart contracts: Issues and solutions. Computer Science Review **43**, 100450 (2022) 195

[34] Protofire: Solhint. https://github.com/protofire/solhint 15, 128, 225

[35] Radford, A., Narasimhan, K.: Improving language understanding by generative pre-training (2018). URL `https://api.semanticscholar.org/CorpusID:49313245` 196

[36] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J.: Exploring the limits of transfer learning with a unified text-to-text transformer. The Journal of Machine Learning Research **21**(1), 5485–5551 (2020) 154, 197

[37] Rogers, A., Kovaleva, O., Rumshisky, A.: A primer in bertology: What we know about how bert works (2020) 205

[38] Sanh, V., Debut, L., Chaumond, J., Wolf, T.: Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. arXiv preprint arXiv:1910.01108 (2019) 146, 154, 164, 196

[39] Service, R.W.: Book review: Corbin, j., & strauss, a.(2008). basics of qualitative research: Techniques and procedures for developing grounded theory . thousand oaks, ca: Sage. Organizational Research Methods **12**(3), 614–617 (2009) 192

[40] SolAI: Sóley: Identification and automated detection of logic vulnerabilities in ethereum smart contracts via large language models (2024). URL `https://figshare.com/s/efa4967b9eeb9dc6ee96`. Accessed: 2024-06-13 193, 228

[41] Solidity Team: Solidity compiler documentation. `https://docs.soliditylang.org` 194, 195, 199, 207, 209

[42] Soud, M., Liebel, G., Hamdaqa, M.: A fly in the ointment: an empirical study on the characteristics of ethereum smart contract code weaknesses. Empirical Software Engineering **29**(1), 13 (2024) 195, 196, 199, 203, 205, 206, 218, 224

[43] Soud, M., Qasse, I., Liebel, G., Hamdaqa, M.: Automesc: Automatic framework for mining and classifying ethereum smart contract vulnerabilities and their fixes. In: 2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 410–417. IEEE (2023) 157, 198, 226

[44] Sun, X., Tu, L., Zhang, J., Cai, J., Li, B., Wang, Y.: Assbert: Active and semi-supervised bert for smart contract vulnerability detection. Journal of Information Security and Applications **73**, 103423 (2023) 16, 204, 225

[45] Sun, Y., Wu, D., Xue, Y., Liu, H., Wang, H., Xu, Z., Xie, X., Liu, Y.: Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pp. 1–13 (2024) 15, 199, 205, 206, 225

[46] Szabo, N.: Smart contracts: building blocks for digital markets. EXTROPY: The Journal of Transhumanist Thought,(16) **18**(2), 28 (1996) 119, 193

[47] Team, V.: Principles and goals. URL `https://docs.vyperlang.org/en/stable/` 194

[48] Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y.: Smartcheck: Static analysis of ethereum smart contracts. In: 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), pp. 9–16. IEEE (2018) 15, 128, 191, 195, 196, 200, 224

[49] Torres, C.F., Schütte, J., et al.: Osiris: Hunting for integer bugs in ethereum smart contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference, pp. 664–676. ACM (2018) 15, 128, 195, 225

[50] Vitalik Buterin, G.W.: Archived ethereum white paper (2014). URL `https://static.peng37.com/ethereum_whitepaper_laptop_3.pdf` 194

[51] Wartschinski, L., Noller, Y., Vogel, T., Kehrer, T., Grunske, L.: Vudenc: vulnerability detection with deep learning on a natural codebase for python. Information and Software Technology **144**, 106809 (2022) 26, 198, 201

[52] Weiser, M.: Program slicing. IEEE Transactions on software engineering (4), 352–357 (1984) 201

[53] Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014) 2, 5, 6, 7, 145, 191, 193, 194

[54] Zhang, P., Xiao, F., Luo, X.: A framework and dataset for bugs in ethereum smart contracts. In: 2020 IEEE international conference on software maintenance and evolution (ICSME), pp. 139–150. IEEE (2020) 3, 13, 14, 15, 53, 54, 62, 63, 64, 65, 75, 90, 92, 119, 121, 122, 134, 135, 178, 199, 203, 205, 206, 213, 224

[55] Zhang, Z., Zhang, B., Xu, W., Lin, Z.: Demystifying exploitable bugs in smart contracts. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 615–627. IEEE (2023) 2, 15, 199, 224, 225

[56] Zhao, J., Chen, X., Yang, G., Shen, Y.: Automatic smart contract comment generation via large language models and in-context learning. Information and Software Technology **168**, 107405 (2024) 146, 192

[57] Zheng, Z., Xie, S., Dai, H.N., Chen, X., Wang, H.: Blockchain challenges and opportunities: A survey. International journal of web and grid services **14**(4), 352–375 (2018) 5, 193

[58] Zhou, Y., Sharma, A.: Automated identification of security issues from commit messages and bug reports. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, pp. 914–919 (2017) 198