Journal Pre-proof

An Ontology of Defects for Ethereum and its Smart Contracts

Michele Pasqua, Sofia Mari , Ferdinando Santoro , Mariano Ceccato

PII: S2096-7209(25)00145-9
DOI: https://doi.org/10.1016/j.bcra.2025.100418
Reference: BCRA 100418

To appear in: *Blockchain: Research and Applications*

Received date: 13 August 2025
Revised date: 26 September 2025
Accepted date: 5 November 2025

Please cite this article as: Michele Pasqua, Sofia Mari , Ferdinando Santoro , Mariano Ceccato , An Ontology of Defects for Ethereum and its Smart Contracts, *Blockchain: Research and Applications* (2025), doi: https://doi.org/10.1016/j.bcra.2025.100418

This is a PDF of an article that has undergone enhancements after acceptance, such as the addition of a cover page and metadata, and formatting for readability. This version will undergo additional copyediting, typesetting and review before it is published in its final form. As such, this version is no longer the Accepted Manuscript, but it is not yet the definitive Version of Record; we are providing this early version to give early visibility of the article. Please note that Elsevier's sharing policy for the Published Journal Article applies to this version, see: https://www.elsevier.com/about/policies-and-standards/sharing#4-published-journal-article. Please also note that, during the production process, errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

© 2025 The Authors. Published by Elsevier Ltd on behalf of Zhejiang University Press.
This is an open access article under the CC BY-NC-ND license
(http://creativecommons.org/licenses/by-nc-nd/4.0/)

Journal Pre-proof

Highlights

An Ontology of Defects for Ethereum and its Smart Contracts

Michele Pasqua, Sofia Mari, Ferdinando Santoro, Mariano Ceccato

- Comprehensive list of Ethereum defects, encompassing vulnerabilities and code flaws, taken from the research literature.
- The Ethereum defects collected come with a description, code examples, and mitigation strategies.
- A tagging system to fine-grain categorize Ethereum defects.
- A visualization tool for easy exploration of Ethereum defects.

Journal Pre-proof

Journal Pre-proof

An Ontology of Defects for Ethereum and its Smart Contracts

Michele Pasqua[a,*], Sofia Mari[a], Ferdinando Santoro[a], Mariano Ceccato[a]

[a] *University of Verona - Dept. of Computer Science, Strada le Grazie, 15, Verona, 37134, Verona, Italy*

Abstract

With Ethereum's rise as the leading platform for decentralized applications, securing Ethereum smart contracts, very often having a financial impact, becomes paramount. Existing research lacks a comprehensive overview of Ethereum defects (and the terminology is often inconsistent), making it difficult for researchers, developers, and industry professionals to navigate this nowadays critical topic. This necessitates a unified source of information detailing defects affecting Ethereum and its smart contracts, along with their root causes, impact, and mitigation strategies.

In this paper, we propose a *knowledge base of defects*, encompassing security vulnerabilities and code flaws found in the Ethereum blockchain and its smart contracts. We started by performing a systematic literature review to identify the currently known defects and then created a *hierarchical tag system* to classify them. This system was then used to build an *ontology* allowing users to easily search and learn about Ethereum defects. We also implemented EDOV, a tool to graphically navigate and explore the ontology, perform search queries, and visualize defect details, such as examples of defective/fixed code. As new defects may appear in the future, the ontology and the tool are built with extensibility in mind.

We believe this research is a valuable contribution to helping developers and practitioners avoid common mistakes, improving the overall security and reliability of the Ethereum ecosystem.

Keywords: Ethereum smart contracts, Vulnerability classification, Code

*Corresponding author
Email address: `michele.pasqua@univr.it` (Michele Pasqua)

Journal Pre-proof

Journal Pre-proof

flaw classification, Solidity language, Defects ontology

1. Introduction

Blockchain technology [1], introduced in 2008, has rapidly revolutionized the IT sector and other related sectors, like finance. Different from traditional centralized systems, blockchain offers a decentralized, transparent, and secure platform for transactions. It eliminates the need for intermediaries, e.g., banks in finance, by utilizing a distributed ledger shared among all users of the system, e.g., yielding the so-called Decentralized Finance, or DeFi [2].

While blockchain's initial application was for cryptocurrencies like Bitcoin [1], its potential extends far beyond finance. Smart contracts, i.e., self-executing programs on the Ethereum blockchain, enable new applications in business, supply chain management, smart industry, and more.

However, the rapid development of smart contract technology has led to security and dependability risks. The peculiarity of smart contracts is that program code and the actions performed (e.g., money transactions) are immutable: once written on the blockchain, smart contracts cannot be modified nor deleted, even in case of programming defects identified after deployment.

For instance, security vulnerabilities in smart contracts can be exploited by malicious users to gain unauthorized access or financial benefits. This is what happened in 2016 with the DAO incident [3], where around $70M were stolen from the Ethereum blockchain. The attack worked as follows, and it exploited a *Re-entrancy* vulnerability on the `withdraw` function of the DAO contract [4]. The function was coded to first send Ether (i.e., the Ethereum's cryptocurrency) to the user and then update the user's balance to reflect the withdrawal. The attacker used a malicious contract that, upon receiving the Ether from the DAO, immediately triggers the fallback function. This function was programmed to call the DAO's `withdraw` function again (thus, to re-enter the contract). Since the balance had not yet been updated (decremented) at the time of re-entry, the DAO contract approves the withdrawal repeatedly in a recursive loop, allowing the attacker to drain funds multiple times within a single transaction.

The result of the attack was the loss of more than $70M, causing the Ethereum community to controversially choose to execute a *hard fork* of the entire network to effectively roll back the chain's history and restore the stolen funds to the original investors. This split the network into Ethereum

Journal Pre-proof

(ETH), which implemented the fork, and Ethereum Classic (ETC), which preserved the original, unforked history.

Another example is a critical vulnerability in the smart contracts provided by Parity Technologies[1], leading to two major incidents in 2017. The vulnerability stemmed from an error in the design of the Parity multi-signature wallet contracts, which used a shared library contract to store core logic and save gas costs.

The first incident was due to an attack exploiting a vulnerability of the `initWallet` function of the multi-sig contract, allowing an attacker to re-initialize the contract and become the sole owner, bypassing the multi-signature requirement. As a result, the attacker was able to steal $31M from three vulnerable wallets. After that attack, Parity deployed a new library contract to fix the issue. However, this new library contract was never properly initialized.

The second incident was due to an attack calling the `initWallet` function on the uninitialized library contract used as a patch for the first attack, making the attacker the owner of the library. Then, the attacker called the `selfdestruct` function, which was a part of the contract's code, effectively disabling the library contract. Since hundreds of multi-sig wallets relied on this shared library for their core operational logic, the disabling of the library instantly rendered all those dependent wallets unusable. As a result, $150M were permanently locked in the wallets, as the contracts could no longer exercise any operation to move the Ether.

Financial loss is not the only concern. DETER [5] was the first discovered attack on the Ethereum node's transaction pool (`txpool`), and it consists of a low-cost Denial-of-Service (DoS) vector. The attack works in three steps. First, it crafts invalid transactions: the attacker sends transactions that are currently invalid (e.g., they have a too-high nonce), making them *future transactions*. This ensures they will never be mined, resulting in a zero Ether cost for the attacker. Second, it forces admission: the attacker sets a very high Gas price on these invalid future transactions. Due to a risky design behavior in many Ethereum client implementations (like Geth), the `txpool` optimistically admits these high-gas-price transactions into the pool. Third, it achieves DoS: once admitted, these high-priority, zero-cost transactions occupy the limited `txpool` space, evicting legitimate, normal

[1] Parity Technologies: `https://www.parity.io`

Journal Pre-proof

transactions (which typically have lower Gas prices).

The result is that the victim node's transaction pool is filled with transactions that can never be mined, presenting a false empty view of available, valid transactions to downstream services like mining pools. This cripples the victim node's ability to participate in mining and transaction propagation, effectively denying service to legitimate users and victimizing the miner by causing them to produce blocks with zero or low revenue.

Another example is NURGLE [6], which was the first discovered attack on the Ethereum storage layer, and it consists of a Denial-of-Service specifically targeting the *Merkle Patricia Trie* (MPT), the data structure used by Ethereum and compatible blockchains to manage storage data (e.g., account balances, contract data). The attack works in two steps. First, it crafts malicious transactions: the attacker submits transactions that create new accounts or storage entries whose `keccak256` hash prefixes are strategically chosen to increase the depth of the MPT. Second, it causes resource exhaustion: the increased depth and complexity of the MPT causes the blockchain to consume significantly more CPU and I/O resources for all subsequent state maintenance and verification, thereby achieving a sustained DoS impacting blockchain performance.

The result is a persistent attack degrading the overall performance of Ethereum or other MPT-based blockchains.

For these reasons, code review, potentially supplemented by automated analysis tools, is indispensable to detect and mitigate smart contract defects before deployment and the irreversible commitment of erroneous transactions to the blockchain. The code review process requires understanding blockchain and smart contract weaknesses, as well as known code flaws and security vulnerabilities. Unfortunately, there is currently a lack of comprehensive resources to inform developers (in building more secure and dependable code) and researchers (in building effective verification and validation tools) about smart contract defects. Indeed, actual sources of information are either incomplete (consider some defects only) or conflicting with each other (in terms of definitions and terminology).

For instance, DASP10 [7] is an open and collaborative project that brings together the nine major security issues in Ethereum smart contracts discovered by the security experts' community. Vulnerabilities are described and accompanied by code examples with references to real-world smart contracts. The SWC [8] is a more fine-grained project, presenting thirty-seven security vulnerabilities. While providing good details, these lists do not represent

a complete collection of known security vulnerabilities (as we will see, the considered vulnerabilities by those projects are a small subset of the actual discovered vulnerabilities), and there are some shared vulnerabilities among them, often referred to by different names. More importantly, these lists only consider smart contract security aspects, overlooking their dependability. Similarly, academic sources are not comprehensive, as they focus either on smart contracts or blockchain infrastructure issues, thus not providing a holistic view of blockchain security and reliability encompassing all layers of the blockchain stack. In addition, a remarkable number of vulnerabilities and defects are not considered in literature surveys. For these reasons, we advocate for a single, comprehensive, and structured *knowledge base of smart contract defects*, supporting developers and researchers in making more secure and dependable blockchain-enabled applications.

As Ethereum [9] is currently the most popular and used blockchain, this research focuses on code flaws and security vulnerabilities, or more generally *defects*, in Ethereum and its smart contracts. In particular, this research aims to address this issue by providing an *ontology of Ethereum defects*, distilled after a systematic review of existing literature on Ethereum. Differently from existing literature surveys, the resulting ontology includes descriptions and code examples of identified defects, encompassing security vulnerabilities and code flaws. Additionally, a categorization system and a graphical visualization tool were developed to facilitate understanding and analysis. Our findings include:

Comprehensive Defect List A detailed and comprehensive list of known Ethereum defects, providing descriptions, code examples, and mitigation strategies;

Categorization System A tagging system to categorize Ethereum defects, enabling better organization and understanding;

Visualization Tool A graphical visualization tool (EDOV) to document Ethereum defects for easy exploration and understanding. The tool also provides query capabilities to search for defect application scenarios and mitigation strategies.

This research provides valuable insights for researchers, developers, and industrialists working with Ethereum smart contracts. By understanding common defects, they can develop more secure and dependable decentralized

Journal Pre-proof

Journal Pre-proof

applications. Indeed, the EDOV tool, and the ontology underneath, can be used to understand how Ethereum defects work and how they can be mitigated by exploiting previously collected knowledge. Developers can use the information provided by the tool to write more secure and dependable smart contracts, while researchers can use it to devise novel detection strategies or mitigation mechanisms.

Synopsis. The paper is organized as follows. Section 2 introduces the reader to Ethereum and its smart contracts, with a brief detour on security vulnerabilities. Section 3 reports the systematic literature review of Ethereum defects, whose results have been used to define the Ethereum defect ontology. The latter is presented and discussed in Section 4. Section 5 describes EDOV, a visualization tool developed to query the Ethereum defect ontology. Finally, Section 6 concludes the paper with a discussion of future work.

2. Background

2.1. Ethereum and Solidity

Ethereum is an open-source platform for decentralized applications, based on blockchain [1] technology. On the Ethereum network, it is possible to write simple programs, called *smart contracts* [10, 11], that (semi-)automatically manage the underlying network cryptocurrency, called *Ether* (ETH). The actions that can be performed in Ethereum are transactions, i.e., transfer of funds or data between different network nodes. Every new transaction is irreversible and it is permanently added in a new block that updates the blockchain [10, 11].

In the Ethereum network, each node has an account identified by an address (a sequence of 20 bytes). There are two types of accounts: *Externally Owned Accounts* (EOA) and *contract accounts* [9]. The former is a simple address that does not point to any code: it can only emit and receive transactions (similarly to Bitcoin wallets [12]). The latter is the identifier of a smart contract deployed in the network, which is run whenever a transaction is sent to its address [11].

Before 2022, Ethereum used a *proof-of-work* (POW) system as a consensus mechanism [13]. Participants of the network, called *miners*, use their computational power and cryptocurrency assets to compete. The miner that succeeds is allowed to add blocks to the Ethereum blockchain and gets a reward [9]. Rewards are paid by the users who invoke the execution of a smart

Journal Pre-proof

```
pragma solidity ^0.6.0;

contract SimpleBank {
   mapping(address => uint256) private balances;

   function deposit(uint256 amount) public
        payable {
      require(msg.value == amount);
      balances[msg.sender] += amount;
   }

   function deposit100() public payable {
      require(msg.value == 100);
      balances[msg.sender] += 100;
   }

   function withdraw(uint256 amount) public {
      require(amount <= balances[msg.sender]);
      balances[msg.sender] -= amount;
      msg.sender.transfer(amount);
   }
}
```

Listing 1: Solidity code example.

```
Runtime Code:
6080604052600436106100345760003560e01c8063140
e9ac714610039 ... 6000206000828254019250508190
55505056fe
Metadata:
a2646970667358221220e62b6e0d256ecbc0a1b39b99b
f0a2b509ed60dd83c71541b2d00fed1bde5a9e464736f
6c634300060b0033
```

Listing 2: EVM bytecode example.

```
PUSH1 0x80 PUSH1 0x40 MSTORE PUSH1 0x4
CALLDATASIZE LT PUSH2 0x34 JUMPI PUSH1 0x0
CALLDATALOAD PUSH1 0xE0 SHR DUP1
PUSH4 0x140E9AC7 EQ PUSH2 0x39 JUMPI
...
PUSH1 0x0 KECCAK256 PUSH1 0x0 DUP3 DUP3
SLOAD ADD SWAP3 POP POP DUP2 SWAP1 SSTORE
POP POP JUMP INVALID
```

Listing 3: EVM Opcodes example.

contract or simply want to transfer funds to other accounts. In fact, every operation in the network has a cost expressed in the unit of *Gas*, and the price per unit is expressed in Wei, a fraction of an Ether. As of today, Ethereum switched to a *proof-of-stake* (POS) consensus mechanism (Ethereum 2.0) [14]. In a POS setting, miners are replaced by *stackers*, which validate transactions based on the amount of cryptocurrency they have staked instead of the computational power they are able to employ.

Solidity Language. Smart contracts for the Ethereum blockchain can be written with different high-level programming languages, but *Solidity* [15] is indubitably the most widespread [16]. Solidity is a Turing-complete object-oriented language, and smart contracts are basically objects with functions and fields.

The example in Listing 1 reports a smart contract written in Solidity to implement a simple banking system. The field `balances` stores the internal state of the smart contract. It is a key-value map that associates every address to an integer value representing the funds owned by the address account. The functions `deposit` and `deposit100` allow the users to deposit currency into their virtual account. The former allows the deposit of an arbitrary amount; the latter is a special case that allows the transfer of exactly 100 Wei. The `withdraw` function allows the user to get back a certain amount of Ether previously deposited. Solidity provides different primitives to interact with the blockchain environment, for instance: `transfer`, that

Journal Pre-proof

sends Ether to a certain address; `revert`, that makes the transaction fail and rolls back to the state preceding the transaction; `require`, that enforces a certain boolean condition and in case the condition is not met a `revert` is performed.

Ethereum Virtual Machine. In order to actually run a smart contract on the Ethereum blockchain, the Solidity source code needs to be compiled into *EVM bytecode*, in order to be executed by the Ethereum Virtual Machine [17].

Given a smart contract, the Solidity official compiler solc generates the *creation code* and the *runtime code*. The former is the constructor of the smart contract, that performs the initial operations and deploys the runtime code on the blockchain (the constructor code is then discarded and not stored in the blockchain [18]). The latter is the actual bytecode deployed on the blockchain, and it is divided into three main segments. The first segment contains the opcodes that the EVM executes; the second segment is optional and contains static data (e.g., strings or constant arrays); the last segment contains the metadata. In particular, the metadata segment contains compilation information, such as the compiler version and the (hashed) sources used, in order to verify its source code [15]. The metadata segment is hashed and appended to the contract bytecode.

In addition, solc also produces the *Application Binary Interface* (ABI), a file containing the list of the functions in the smart contract that can be called by a user, together with the type and number of parameters. Functions are not identified by their name but by the hash of their signature. The ABI file is not deployed on the blockchain, it is distributed separately to all parties that aim to interact with the smart contract.

The EVM bytecode is technically composed of *EVM opcodes* that can be grouped into categories, including arithmetic and logic operations, control flow operations, stack operations, environmental and block information, memory and storage operations, and system operations. The complete list of opcodes with their semantics is defined in the Ethereum yellow paper [19], and there can be little variations among different EVM versions. Listing 2 shows a portion of a Solidity smart contract compiled into EVM bytecode, while Listing 3 shows the translation of the bytes into EVM opcodes. Bytecode can be easily parsed into opcodes, which are the minimum instructions that the EVM can execute and are identified with bytes.

Ethereum Runtime Architecture. As summarized by Chen et al. [20], the Ethereum blockchain runtime is a software stack composed of four layers,

Journal Pre-proof

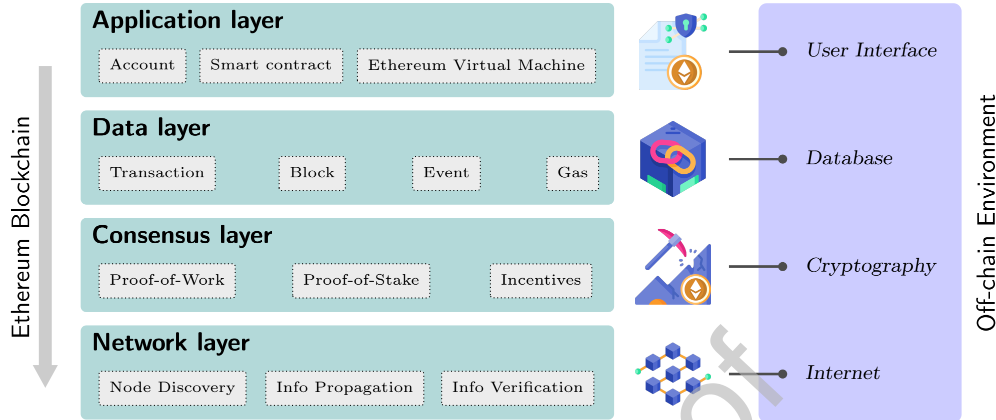


Figure 1: Architecture of the Ethereum blockchain.

as depicted in Figure 1. The application layer allows Ethereum clients to execute smart contracts in the EVM; the data layer contains the blockchain data structures; the consensus layer ensures consistency for the network state; and the network layer controls the peer-to-peer (P2P) communication so that nodes are constantly updated about the blockchain state.

In particular, in the *Application layer* resides smart contracts and their business logic, together with EOA addresses. A contract account or EOA has a dynamic state, defined by a *nonce*, which tracks the number of transactions initiated by the EOA owner or the number of contracts created by the contract account; a *balance*, which records the amount of Wei owned by the EOA or contract account; a *storageRoot*, which is the hash of the root of the storage data structure recording the state variables of a contract account; and a *codeHash*, which is the bytecode hash value of a contract account. The state of the blockchain is defined as the state of all its accounts.

The *Data layer* manages Ethereum transactions, that is, interactions between an EOA (called sender) and another EOA or contract account (called recipient). A transaction is specified by a *nonce*, which is a counter keeping track of the total number of transactions initiated by the sender; a *recipient*, which specifies the destination EOA or contract account of the transaction; a *value*, which is the amount of money (in Wei) to be transferred from the sender to the recipient (if applicable); an *input*, which is the bytecode or data sent alongside the transaction; a *gasPrice* and a *gasLimit*, which specify the unit price and the maximum quantity of gas that the sender is willing to

Journal Pre-proof

pay to the winning miner of a block containing the transaction; and a *signature*, which is the ECDSA encrypted signature of the sender. Executing a transaction updates the state of the accounts involved.

The *Consensus layer* involves the mining and consensus protocol. In Ethereum, multiple miners could create valid blocks at the same time. A variant of the GHOST consensus protocol is used to select the "heaviest" branch as the main chain, where the "heaviest" branch is the subtree rooted at the fork in question and having the highest cumulative block difficulty.

Finally, the *Network layer* involves node low-level communication. Specifically, Ethereum is a structured P2P network in which each node (i.e., clients) stores a copy of the entire blockchain. For the purpose of node discovery and routing, each node maintains a dynamic routing table. Ethereum uses the RLPx protocol to discover target clients and uses the Ethereum Wire protocol to facilitate the exchange of information (i.e., transactions and blocks) between clients.

2.2. Decentralized Applications

As natural application of smart contracts, we can find *Decentralized applications* (Dapps). These applications are created for decentralized networks and combine a smart contract, used as the backend, with a user interface that constitutes the frontend. The main innovation that Dapps introduce is the fact that the backend runs on a decentralized P2P network (a blockchain) and not on a central server. At present, these decentralized applications are used in various fields such as finance, gambling, governance, asset exchange, e-voting, healthcare, logistics, IoT solutions, and trading [21].

As an example, smart contracts introduced a revolution in the financial world, moving financial activities to a decentralized setting. Indeed, *Decentralized Finance* (DeFi) [2] aims to mimic on the blockchain a wide range of financial services, making them more efficient, transparent, and inclusive [22]. Decentralization makes it possible to create a more secure and censorship-resistant financial ecosystem. In addition, transparency ensures a high level of trust and verifiability. Among the main DeFi applications, we can find *Decentralized Exchanges* (DeXs), which allow users to exchange digital assets directly with each other without intermediaries making transactions faster. Other pillars of DeFi are trading, decentralized insurance, and stablecoins, which are cryptocurrencies pegged to the value of a real (non-digital) asset [23].

Journal Pre-proof

Journal Pre-proof

Some Dapps issue their own cryptocurrency, which is generally called *token*. Tokens are digital assets internally managed by one or more smart contracts, that can be ultimately exchanged for Ether. Each token, hence, has its own pricing, which can fluctuate over time as fiat money, and can be exchanged with other tokens. Tokens can be used by third-party applications (e.g., wallets and exchangers) when the smart contract issuing them adheres to standard protocols specifically developed for token creation and exchange. Among the most popular standards, we can find the following.

ERC20 This is the most popular standard for developing *fungible tokens* on Ethereum [24]. Fungible means that a token is identical and interchangeable with any other ERC20 token of the same type.

ERC721 This is the most popular standard for developing *non-fungible tokens* (NFTs) on Ethereum [24]. Non-fungible means that a token is unique, indivisible, and irreplaceable, and it represents a one-of-a-kind digital asset.

2.3. Smart Contracts Vulnerabilities

As with any other software, smart contracts are prone to programming errors. These may lead to *security vulnerabilities* in the code deployed on the blockchain, potentially hindering the reliability and security of Dapps. This is particularly problematic in Ethereum since blockchain-based applications are, by design, public, decentralized, and immutable. Defects in smart contracts cannot be patched after deployment, and the (possibly vulnerable) code of a smart contract is publicly available to attackers who may ultimately find a way to exploit a vulnerability, if present. In addition, smart contracts usually deal with profitable digital assets, resulting in an appealing target for cyberattacks [3].

A Prominent Example: Reentrancy. One of the most dangerous vulnerabilities in Solidity (and, hence, in Ethereum smart contracts) consists in the mishandling of possibly reentrant code. It has been made famous due to the catastrophic DAO incident [4], which caused the loss of a large amount of money and serious consequences to the whole Ethereum network. This vulnerability consists of reentering a paying function multiple times while the contract is in an inconsistent state, thus causing a possible leak of funds [25]. In the general case, the "reentrance" exploits the fact that the vulnerable contract calls primitives, such as money transfer, that the malicious contract

Journal Pre-proof

Journal Pre-proof

```
pragma solidity ^0.5.0;

contract ReentrantContract {
   mapping (address => uint) private
       balances;

   function withdraw (uint amount) public {
      require(amount <= balances[msg.sender
          ]);
      if (msg.sender.call.value(amount)()) {
         balances[msg.sender] -= amount;
      }
   }
}
```

Listing 4: A Solidity smart contract vulnerable to Reentrancy.

```
pragma solidity ^0.5.0;

contract MaliciousContract {
   ReentrantContract reentrantContract;

   function attack () public {
      reentrantContract.withdraw(100);
   }
   function () payable {
      reentrantContract.withdraw(100);
   }
}
```

Listing 5: A Solidity smart contract exploiting the Reentrancy vulnerability of Listing 4.

can redefine in such a way as to reenter the vulnerable contract. If a money transfer occurs at each iteration of this loop, the process can be repeated and used to drain all resources from the attacked contract. Indeed, Reentrancy is a consequence of an abuse of dynamicity in Solidity: the semantics of money transfer is dynamic and can be redefined.

A simpler, yet quite common, programming error that may lead to Reentrancy attacks consists in updating the contract state after (instead of before) executing a fund send primitive (i.e., a `call`). An example of a Reentrancy vulnerable contract following this pattern is shown in Listing 4, where the `call` statement at line 7 precedes the update of the variable `balances` at line 8. In Listing 5 we have a (malicious) contract that exploits the Reentrancy vulnerability of the `ReentrantContract`. In particular, the attack is launched by calling the `withdraw` method on the vulnerable contract (line 6). When the `call` primitive is executed in the `withdraw` method, performing the money transfer, the fallback function of the contract `MaliciousContract` is fired. The latter contains a recursive call to the vulnerable contract, again on the `withdraw` method (this is the reentrant code). Since the `balances` variable in the vulnerable contract is updated after the money transfer, the second `withdraw` can be legitimately called, and a second money transfer is performed. The process is repeated until all money is drained from the vulnerable contract.

3. Systematic Literature Review

To distill an ontology of Ethereum defects, which will be presented in Section 4, we leverage the information actually provided by the research literature. In particular, we carried out a *systematic literature review* (SLR)

Journal Pre-proof

of research work concerning smart contract defects. By analyzing the results of the SLR we were able to classify and attach semantic information to the defects actually affecting Ethereum smart contracts.

Despite several surveys on Ethereum security being present in the literature, none of them extensively investigates the problem we aim to tackle with our ontology, thus motivating the SLR presented in this work. Most of the surveys, which will be extensively described in the next subsection, focus on vulnerability detection tools, lacking a comprehensive investigation of vulnerabilities. Moreover, the majority of surveys consider security vulnerabilities only, giving very little space for smart contract defects not related to security aspects. Another issue is that previous studies consider a few vulnerabilities compared to the actual discovered vulnerabilities. The largest previous study [20] comprises 40 vulnerabilities, while, as we will see later in the section, our SLR found 98 vulnerabilities (in addition to 37 code flaws). Moreover, previous studies focus on specific aspects, such as a specific Ethereum layer or the vulnerabilities detected by state-of-the-art analysis tools, while we advocate for a more holistic view of Ethereum security and reliability, encompassing all layers of the Ethereum stack. Another problem overlooked by previous work is defect terminology. The latter is inconsistent in community resources, research papers, and literature surveys. This issue, which we tackle in our work, poses significant problems to developers in accessing the defect knowledge when securing their smart contracts.

3.1. State of the Art

Concerning academic literature surveys, Chen et al. [20] present a comprehensive and systematic investigation of Ethereum's security, including vulnerabilities, attack vectors, and mitigation strategies. The authors discuss 40 types of vulnerabilities based on the layers of the Ethereum architecture and describe the history, cause, tactics, and direct impact of 29 attacks. Regarding countermeasures, the authors enumerate 51 defense mechanisms, dividing them into proactive and reactive, and provide best practices to guide contract development. However, the vulnerabilities illustrated are not comprehensively described and sometimes lack code examples. Moreover, they mainly focus on smart contract security aspects.

Chen et al. [26] define 20 types of smart contract defects and classify them based on whether they hinder security, availability, performance, maintainability, and reusability. Such defects have been manually identified in 587 real-world smart contracts.

Kushwaha et al. [27] present a systematic review of Ethereum security that includes smart contract vulnerabilities, detection tools, real-life attacks, and prevention mechanisms. The 23 vulnerabilities discussed are classified into 3 groups based on their root causes.

Li et al. [28] summarize and analyze DeFi-related vulnerabilities at different layers of the Ethereum blockchain architecture. They then discuss real-world DeFi attacks based on vulnerability-related principles.

Ghosh et al. [29] discuss a detailed blockchain taxonomy, covering key cryptocurrency-related platforms, such as Hyperledger and Multichain, and presenting vulnerabilities related to recent attacks on Bitcoin and Ethereum.

Chu et al. [30] explore smart contract security from the perspective of vulnerability data sources, vulnerability detection, and defense strategies. They investigate existing vulnerability classification frameworks and describe 12 vulnerabilities by grouping them into 3 threat levels.

Compared to the previously mentioned literature reviews, in this work, we provide a more complete and comprehensive analysis of the security and reliability problems of the Ethereum platform. Previous work focuses either on smart contracts or infrastructure issues, while the current work provides a holistic view of Ethereum security and reliability, encompassing all layers of the Ethereum stack. This results in an extensive classification and an in-depth analysis of currently known Ethereum defects, placing particular emphasis on accurate descriptions, explanatory code snippets, consistent terminology, root causes, and best practices or mitigation patterns. In addition, a remarkable number of vulnerabilities and defects were left out of the above-mentioned surveys, which are indeed collected and analyzed in the current work. Moreover, our categorization is more fine-grained than previous studies, having, as we will see later in the section, a four-level hierarchical structure with 18 leaf categories (in addition to topic tags). This provides a comprehensive analysis of Ethereum defects, comprising information on defect scope (where they can take place), root causes (why they take place), and possible exploitation vectors (how they can be exercised).

To help developers mitigate security issues in smart contracts, the security experts' community started to classify known vulnerabilities peculiar to blockchain-based applications. The most important projects in this context consist of the *Decentralised Application Security Project top 10* (DASP10) [7] and the *Smart contract Weakness Classification* (SWC) [8].

DASP10 is an open and collaborative project that brings together the major security issues in Ethereum smart contracts discovered by the security

experts' community. This collection presents the nine most common security vulnerabilities, with the tenth item on the list reserved for future flaws not yet discovered. In DASP10, the vulnerabilities are described and accompanied by code examples with references to real-world smart contracts.

SWC is more fine-grained, presenting 37 security vulnerabilities. As in the case of DASP10, many items in this list are accompanied by a description and code examples with references to real-world smart contracts. The code examples often present both the vulnerable and the patched version.

While providing good details, these lists do not represent a complete collection of known security vulnerabilities, and there are some shared vulnerabilities among them, often referred to by different names. The SLR carried out in this work, on the other hand, presents a more comprehensive and accurate collection and classification of generic smart contract defects (not restricted to security issues), as well as an attempt to unify terminology.

Practitioners have also started to collect information on smart contract vulnerabilities. For instance, DeFi Hack Labs[2] is an educational, community-driven resource for understanding and analyzing past Decentralized Finance (DeFi) hacking incidents. The resource focuses on vulnerabilities affecting DeFi applications only, with references to the smart contracts affected. Still, the project does not provide defect descriptions, patches, and it does not categorize such defects.

Different from all previous sources, with this work, we provide a formalization of the collected knowledge in a machine-readable and extensible ontology. In addition, the developed tool (EDOV) eases the accessibility of the Ethereum defects collected, to benefit the whole Ethereum community.

3.2. Review Methodology

The systematic literature review was performed following the consolidated guidelines proposed by Kitchenham and Charters [31]. The steps carried out during the review will be extensively described in the following paragraphs.

Research Questions. Although the evolution of Ethereum blockchain has been remarkable in recent years, smart contracts still suffer from security vulnerabilities and code flaws, which lack comprehensive investigation. What is also lacking is uniformity in terminology, with defects describing the same issue but using different terms. Moreover, many defects could be grouped

[2]`https://github.com/SunWeb3Sec/DeFiHackLabs`

Journal Pre-proof

Journal Pre-proof

into subsets with the same characteristics. We advocate the need to combine defects with semantic tags, descriptions, and code fragments in order to improve defect understanding.

The present literature review focuses on the security and reliability of Ethereum smart contracts, guided by the following research questions.

RQ1 What are the known defects present in smart contracts?

RQ2 Have these defects been adequately classified and described?

RQ3 Which defects have been comprehensively studied and come with a mitigation strategy?

RQ4 Is the terminology of defects uniform across current sources of information?

RQ5 Which datasets of defective smart contracts are publicly available?

Information Sources. To answer the above research questions, we exploited the information coming from three different databases of research articles. Specifically, we used as information sources the following reputed search engines: IEEE Xplore, Science Direct, and ACM Digital Library. Following the guidelines proposed in [31], a search strategy was devised for the purpose of answering the previously mentioned research questions. As for the academic field, material was identified from major conferences and journals. The quality of scholarly publications was assessed based on rankings compiled by Computing Research and Education (CORE) conference ranking and SCImago Journal Rank (SJR). Finally, since the academic literature may not contain all relevant material (e.g., code snippets and public datasets of smart contracts), a web search engine (i.e., Google search) was used to enrich the search, with the relevance and reliability of findings assessed through the authors' domain knowledge.

Search Queries. Search queries have been created after selecting a set of keywords modeling the core aspects of our investigation. Possible singular/plural declinations and differences in formatting conventions have been considered, resulting in the following list of keywords.

Journal Pre-proof

`DApp`	{"DApp", "Dapp", "dapp"}
`Smart-contract`	{"Smart-contract", "Smart-contracts", "Smart contract", "Smart contracts", "smart-contract", "smart-contracts", "smart contract", "smart contracts"}
`Solidity`	{"Solidity", "solidity"}
`Ethereum`	{"Ethereum", "ethereum"}
`Survey`	{"Survey", "survey"}
`Vulnerability`	{"Vulnerability", "Vulnerabilities", "vulnerability", "vulnerabilities"}
`Defect`	{"Defect", "Defects", "defect", "defects"}
`Bug`	{"Bug", "Bugs", "bug", "bugs"}

We considered both `Vulnerability` and `Defect` since the line between vulnerabilities and defects in the literature is very blurred, and, often, these two terms are used interchangeably as synonyms. We devise this ambiguity as a bad practice, for which we propose a solution with our ontology.

The search queries executed on the three databases are semantically the same, even if they have syntactic differences due to the different interfaces provided by the search engines. Building upon the previously identified keywords, we feed each search engine with queries adhering to the following pattern (queries have been searched in documents abstract and title):

(`DApp` **or** `Smart-contract` **or** `Solidity` **or** `Ethereum` **or** `Survey`) **and** (`Vulnerability` **or** `Defect` **or** `Bug`)

Eligibility Criteria. After carrying out an initial selection of documents based on the above-mentioned search queries, some inclusion and exclusion criteria have been defined to obtain only the articles deemed relevant for the purposes of the SLR, excluding all those that did not satisfy the criteria defined in Table 1. We included only papers published in reputed venues, using established journal and conference rankings such as SCImago Journal Rank [32] (journals in the first quartile Q1) and CORE conference ranking [33] (conference with grade A or A*). We included only papers related to vulnerabilities, defects, or datasets connected to the Ethereum blockchain or its smart contracts by applying a two-step qualitative assessment (explained below). We excluded papers not written in English and published before 2019.

Search Results. After querying the three databases, we obtained a total of 871 documents. We then applied the inclusion and exclusion criteria to screen

Journal Pre-proof

Table 1: Eligibility criteria.

Inclusion Criteria	Exclusion Criteria
Grade A/A* conference or Q1 journal article	Non-English article
Description of vulnerabilities or defects related to Ethereum smart contracts or blockchain	Published before 2019
Reference to public datasets of smart contracts annotated with vulnerabilities or defects	Lack of focus on vulnerabilities and defects

articles for eligibility (Figure 2 reports the number of documents resulting from each screening step). After dropping the articles published before 2019, we ended up with 779 documents, which is quite close to the initial number of search results. This highlights the fact that the problem has been investigated mainly in recent years. After selecting the documents published in top-tier research journals and conference proceedings, the set of articles amounted to 356 samples, which decreased to 339 after some duplicate removal.

The next step involved the review of document abstracts to select only the articles relevant to the SLR objective based on the inclusion criteria. Next, the remaining 71 articles were subjected to a more detailed analysis of the whole document to ensure that they indeed met the inclusion and exclusion criteria. At the end of the pipeline, 21 articles were retained.

3.3. Analysis of the Systematic Literature Review Results

After a careful, in-depth analysis of the resulting 21 articles, we retrieved a total of 135 unique Ethereum defects. During the analysis, we noted that many defects have common characteristics and could be grouped together by using tags. Hence, we devised a *semantic tag system* based on the knowledge acquired during the SLR with the help of three experienced smart contract developers. These tags will be used to classify the defects, as basic blocks of the ontology we will present in Section 4. We collaboratively distilled the tags by using card sorting [34], a consolidated methodology for understanding how people organize information and for developing logical structures (e.g., taxonomies) from data. Card sorting helps to uncover underlying relationships and identify common themes among items, leading to a more abstract and intuitive organization. The names of the 135 defects have been printed on cardboards (with brief descriptions), and all participants have been asked to group them guided by three defect dimensions: where, why, and how. The first dimension models *where* the defects take place, w.r.t. the Ethereum

Journal Pre-proof

architecture. In other words, we ask in which layer of the Ethereum stack we can identify a defect. The second dimension models *why* the defects take place. In other words, we ask what the root cause of a defect is. The third dimension models *how* the defects take place. In other words, we ask which exploitation vector actually induces a defect. After hours of brainstorming, we ended up with 28 tags, to classify smart contract defects.

Defect Tag System. Tags have been created by us with the help of three experienced smart contract developers by exploiting the knowledge accumulated in the SLR. The tag system has been developed by analysing defects under three dimensions: where, why, and how. The first dimension models where the defects take place, w.r.t. the Ethereum architecture, resulting in the following 4 *structural tags*.

Structural Tags

Application Refers to the Ethereum Application layer, where clients execute smart contracts in the EVM and where contracts are associated with Ethereum accounts.

Data Refers to the Ethereum Data layer, involving the blockchain core data structures.

Consensus Refers to the Ethereum Consensus layer, ensuring a consistent state for the whole blockchain.

Network Refers to the Ethereum Network layer, managing a peer-to-peer network of clients such that a node can always get the updated state of the blockchain from active nodes.

The second dimension models why the defects take place, resulting in the following 3 *root tags*.

Root Tags

Implementation Refers to implementation logic bugs (e.g., overflow).

Inconsistency Refers to inconsistencies and incompatibilities related to the use of different compiler versions, deprecated functions, constructor errors, typos, inheritance, and external calls and data types (e.g., no return value in ERC20 interfaces).

Auth Refers to excessive permissions (e.g., to change a contract owner).

The third dimension models how the defects take place, resulting in the following 18 *exploit tags*.

Journal Pre-proof

Journal Pre-proof

Exploit Tags

Library Misuse Refers to problems that occur both in the development and use of libraries.

Signature Refers to defects due to improper use of method signatures or their unexpected modification.

Too much code Refers to defects due to the use of more code than what is actually needed.

Not Enough Code Refers to defects due to the presence of less code than what is actually needed.

Casting Refers to defects due to incorrect casting of variables, both implicit and explicit.

Misconfiguration Refers to defects due to an incorrect configuration of the environment (e.g., incorrect version of the compiler).

Gas-Related Refers to an incorrect gas usage.

Logical Flaw Refers to defects due to the business logic of a smart contract (e.g., to gain an unfair financial profit).

Syntactic Error Refers to defects due to typing errors (e.g., the use of wrong constructor names).

Bad Pattern Refers to defects hindering code quality and maintainability.

Identification Refers to weak or easily manipulable methods for verifying user identity or authority.

Access Control Refers to the incorrect implementation of mechanisms to restrict who can execute functions or modify critical state variables.

Randomness/Time Refers to defects due to the improper use of randomness primitives and time manipulation functions.

Overflow Refers to defects due to the improper check of variable overflows.

DoS Refers to defects leading to Denial-of-Service issues in smart contracts.

Encoding Issue Refers to the improper encoding of data (e.g., using short addresses).

External Call Refers to defects due to the improper use of calls to external contract methods.

Memory Error Refers to the improper use of memory and storage variables (e.g., leading to call stack saturation).

Journal Pre-proof

Journal Pre-proof

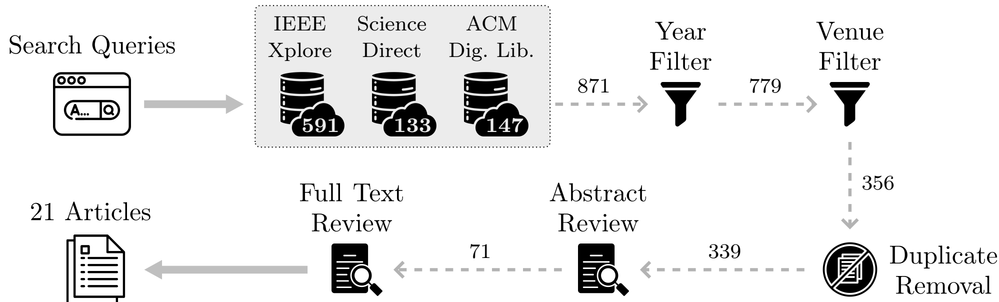


Figure 2: Number of articles included and excluded during the search process.

In addition to such dimensions, we reserved some tags for specific topics that are orthogonal to the tags previously mentioned. Hence, we have the following 3 *topic tags*.

Topic Tags

ERC20 Refers to defects affecting ERC20 tokens (i.e., fungible).

ERC721 Refers to defects affecting ERC721 tokens (i.e., non-fungible).

x.y.z Refers to the version of Solidity in which a defect has been mitigated.

As a final remark, we point out that our classification is orthogonal to the Web3Sec Tagging System[3], which focuses on the severity of a security vulnerability, and it can be incorporated in our ontology.

Answer to Research Questions. To answer **RQ1**, we collected a list of 135 unique Ethereum defects, encompassing security vulnerabilities and code flaws. This defect list (that we report in extension in Appendix A.1) represents the most updated and comprehensive source of information about Ethereum defects in the literature.

To answer **RQ2**, during the reading of academic material, we noted that several papers aims at describing the vulnerabilities and code flaws not specifically for Ethereum, but of blockchain in general. Moreover, the introduction of the defects is often preparatory to the description of tools for analysing or testing smart contracts, so they are in general not sufficiently described.

Furthermore, except for the most well-known defects, descriptions often lack example code fragments, which are very useful for understanding the

[3]https://www.web3sec.news

Journal Pre-proof

Journal Pre-proof

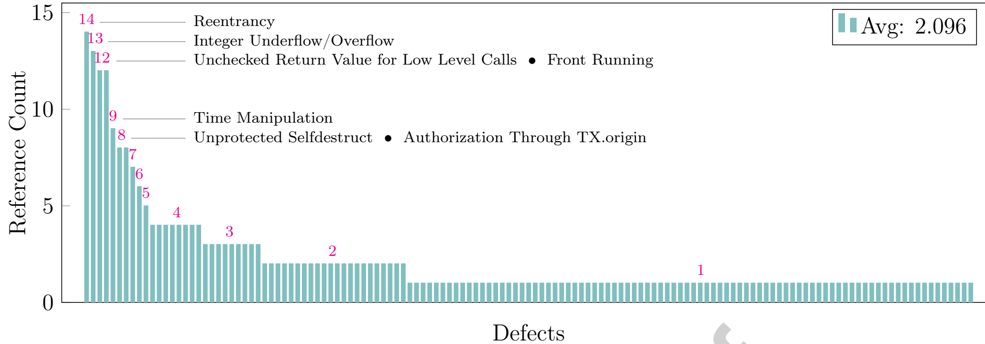


Figure 3: Number of articles referencing the collected defects.

defect. Indeed, it was sometimes necessary to integrate descriptions with other non-academic information sources. This allowed, where possible, to report not only examples of defective code, but also the related patch.

To answer **RQ3**, we can refer to Figure 3 which highlights, for each defect, the number of articles in which that defect was referenced. Among the selected material, the average number of articles referenced a defect is 2.096. Based on this result, we decided that a defect is considered understudied if it has two or fewer citations among the selected articles. Looking at Figure 3, we can see that there are many defects that are little discussed or known. On the contrary, among those most described there are several included in DASP10 or SWC, specifically Reentrancy (the most discussed), Integer Underflow/Overflow, Unchecked Return Values for Low Level Calls, Front Running, Time Manipulation, Unprotected Selfdestruct, and Authorization Through TX.origin.

With regard to the introduction of patches directly by the Solidity compiler, we noted that only 14 out of 135 defects have mitigation from a given compiler version.

For the sake of completeness, a table is provided in Appendix A.4, which gives the reference count for each defect found. In the table, it is also possible to find the count of datasets in which a defect is contained.

In response to **RQ4**, it must be said that during the review process, in some cases, we found that the same phenomenology is referred to by different terms. In general, names differ by a few words, or synonyms are used. However, in some cases, names are completely different, and only after a careful reading of the descriptions or code examinations is it possible to state that those names refer to the same defect. As an example, the names

Journal Pre-proof

DoS with Block Gas Limit and DoS with Unbounded Operations refer to the same code flaw, but it is not immediately obvious that such equivalence. In Appendix A.2 we report all collected defects with the corresponding aliases.

To answer **RQ5**, we annotated each smart contract dataset resulting from the reading of academic articles with the defective smart contracts it contains. This information was enriched through Google searches to increase the number of smart contract samples. It is possible to state that there is some sort of correlation between the defects in the annotated datasets and those that were indicated as the most studied in the previous research questions (e.g., Reentrancy). The majority of the smart contracts came from the Etherscan [16] platform, which is a blockchain explorer for the Ethereum network that allows free searches for transactions, blocks, wallet addresses, smart contracts and other data on the chain. The platform allows developers to validate their smart contracts by submitting the source code and verifying it to add credibility to their contracts. This verification process helps to increase the source code set of smart contracts. However, these smart contracts are not labeled in terms of defective functions, which is why they are often submitted to analysis tools and the results yield collections of annotated smart contracts with their defects. The list of datasets containing defective smart contracts is reported in Appendix A.3.

3.4. Threats to Validity

Threats to the *internal validity* of the work concern the processes of identification, selection, and synthesis of the primary studies. Indeed, the work may be biased by source selection, search queries, and inclusion/exclusion criteria. To mitigate these threats, we followed a consolidated methodology to perform the literature review, guided by the guidelines proposed by Kitchenham and Charters in 2007. This includes the adoption of commonly used inclusion/exclusion criteria (e.g., high-ranked conference publications) and reputed publication databases (e.g., IEEE Xplore).

Threats to the *construction validity* of the study concern the manual development of the tag systems and the defect categorization. We mitigate these threats by involving three experienced smart contract developers in the process. Moreover, we followed a consolidated methodology for understanding how people organize information, which is card sorting. The whole process has been performed by two of the authors and the three external collaborators, supported by a third author to help reach consensus in cases of disagreement.

Journal Pre-proof

Threats to the *external validity* of the study relate to the extent to which the findings of the literature review and the defect classification can be generalized or applied to other settings. We do not claim any generality of the findings outside the context of Ethereum. Moreover, the findings represent the current status of the Ethereum defects, which may change over time.

4. Ethereum Defect Ontology

Looking at the results of the literature review, it appears evident that the Ethereum ecosystem and the research community lack uniformity of terminology about Ethereum defects. There is also no agreement on defect definitions, not even informally.

To shed light on the problem, we start from the roots by providing a clear definition of (security) vulnerability. Indeed, the terms vulnerability and defect are very often used interchangeably in the literature, but we think security-related issues should be considered separately from security-unrelated ones. Such definition, and the next one modeling code flaws, have been validated and refined with the help of the three experienced smart contract developers who participated in tagging the 135 defects.

Definition 4.1 (Vulnerability)**.** An Ethereum *vulnerability* refers to a weakness in smart contracts that can be exploited by an attacker to gain unauthorized access or cause damage.

In the definition, damage is considered very broadly, and it could span from financial loss to service unavailability and data tampering. Some examples of vulnerability are: DoS with Block Stuffing, which may preclude miners from certifying some transactions; Reentrancy, which may lead to the stealing of Ether; and GetToken Anyone, which may lead to an unauthorized ERC20 token transfer.

Dual to vulnerabilities, we provide a definition of non-security issues that we call code flaws.

Definition 4.2 (Code Flaw)**.** An Ethereum *code flaw* refers to any type of non-conformance to a specification or requirement in smart contracts, preventing them from meeting their intended purpose or decreasing their quality.

The key difference between a vulnerability and a code flaw is that the former necessitates an attacker who purposely exploits a weakness in the

Journal Pre-proof

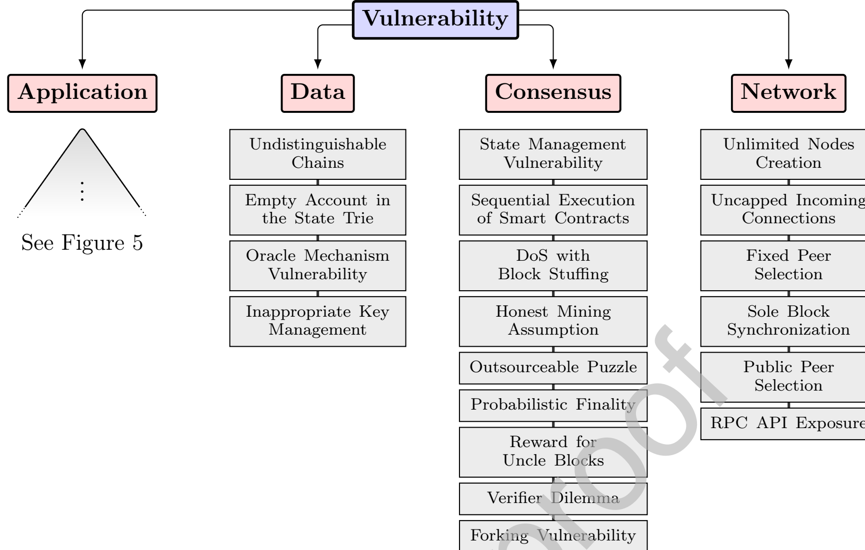


Figure 4: Final classification of vulnerabilities.

system. The second distinction is that code defects, differently from vulnerabilities, may not yield damage. An example is bad programming patterns that do not cause damage but hinder code comprehension and maintainability. Some examples of code flaws are: Gas Costly Pattern, which may lead to a high gas consumption; Missing Return Statement, which may lead to unpredictable EVM behaviors; and BatchTransfer Overflow, which may lead to the incorrect distribution of ERC20 tokens.

Vulnerabilities and code flaws are the root concepts of our Ethereum defect ontology that we will detail in the next subsection. In Subsection 4.2, we will provide some detailed examples of vulnerabilities and code flaws we encountered in the SLR and described in our ontology.

4.1. Defect Classification

The results of our SLR revealed the presence of 135 unique Ethereum defects. Many of them share common peculiarities, which we distilled in the tag system provided in Subsection 3.3, hence, they can be grouped into categories. Indeed, classification is a natural process humans tend to apply in order to simplify a problem. We then classified Ethereum defects into an ontology guided by the tags we extrapolated from the SLR.

As mentioned at the beginning of the section, the first distinction we

Journal Pre-proof

Journal Pre-proof

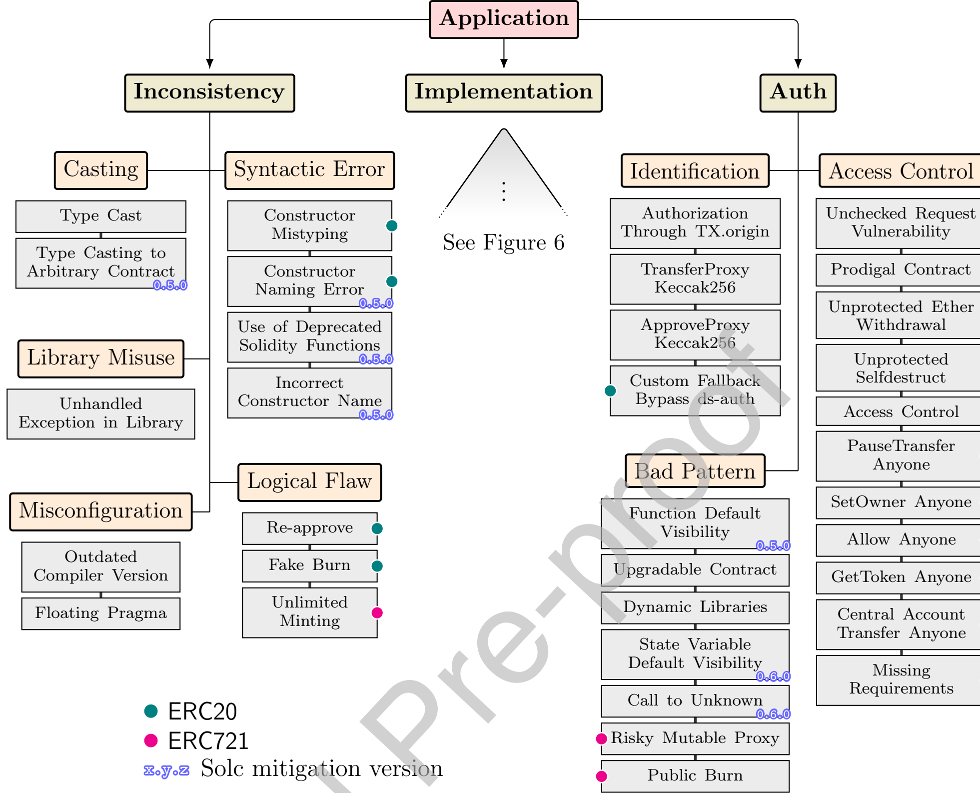


Figure 5: Final classification of vulnerabilities (focus on the Ethereum Application layer).

make is between security-related and security-unrelated defects. Then, the classification process applies the strategy adopted when designing the tag systems based on the where, why, and how dimensions.

Vulnerability classification. Among the 135 defects found, 98 of them have been classified as vulnerabilities, as per Definition 4.1. Then, we started grouping vulnerabilities under the where dimension, that is, the layer of the Ethereum stack they affect. As we can see from Figure 4, 6 vulnerabilities pertain to the Network layer, corresponding to the tag Network, 9 vulnerabilities pertain to the Consensus layer, corresponding to the tag Consensus, 4 vulnerabilities pertain to the Data layer, corresponding to the tag Data, and 79 vulnerabilities pertain to the Application layer, corresponding to the tag Application. Due to the large number of vulnerabilities belonging to the Application layer, the list of such vulnerabilities is expanded in Figure 5.

Journal Pre-proof

We then discriminated vulnerabilities based on their root cause (the why dimension). The root tags we individuated in the SLR all pertain to the Application layer, so no further categorization has been provided for the other Ethereum layers. We have three root causes yielding to vulnerabilities: bugs in the implementation logic, tag Implementation; code inconsistencies, tag Inconsistency; and excessive permission, tag Auth. We inserted 45 defects in the first category, 12 vulnerabilities in the second, and 22 vulnerabilities in the third. Figure 5 depicts vulnerabilities grouped by root tag. Due to the large number of vulnerabilities concerning bugs in the business logic implementation, the list of such vulnerabilities is expanded in Figure 6.

Considering the attack vector (the how dimension), we noted that the following exploit tags can be applied to vulnerabilities: 6 vulnerabilities fall into Library Misuse, specifically 1 belonging to Inconsistency and 5 to Implementation; 4 vulnerabilities fall into Signature, all belonging to Implementation; 1 vulnerability falls into Not Enough Code, belonging to Implementation; 2 vulnerabilities fall into Misconfiguration, both belonging to Inconsistency; 2 vulnerabilities fall into Casting, both belonging to Inconsistency; 7 vulnerabilities fall into Memory Error, all belonging to Implementation; 6 vulnerabilities fall into External Call, all belonging to Implementation; 3 vulnerabilities fall into Encoding Issue, all belonging to Implementation; 4 vulnerabilities fall into DoS, all belonging to Implementation; 8 vulnerabilities fall into Overflow, all belonging to Implementation; 3 vulnerabilities fall into Randomness/Time, all belonging to Implementation; 11 vulnerabilities fall into Access Control, all belonging to Auth; 5 vulnerabilities fall into Identification, specifically 1 belonging to Implementation and 4 to Auth; 6 vulnerabilities fall into Logical Flaw, specifically 3 belonging to Implementation and 3 to Auth; 4 vulnerabilities fall into Syntactic Error, all belonging to Inconsistency; and 7 vulnerabilities fall into Badd Pattern, all belonging to Auth.

Orthogonally, we can classify vulnerabilities in terms of topic tags. Indeed, 20 vulnerabilities are related to ERC-20 tokens, tagged with ERC20 (4 belonging to Inconsistency, 7 to Auth, and 9 to Implementation); 5 vulnerabilities are related to ERC-721 tokens, tagged with ERC721 (1 belonging to Inconsistency; 2 belonging to Auth; and 1 to Implementation); while 8 vulnerabilities have a mitigation version, tagged x.y.z.

Code flaw classification. Among the 135 defects found, 37 of them have been classified as code flaws, as per Definition 4.2. Concerning the where dimension, we noted that all code flaws we found pertain to the Ethereum

Journal Pre-proof

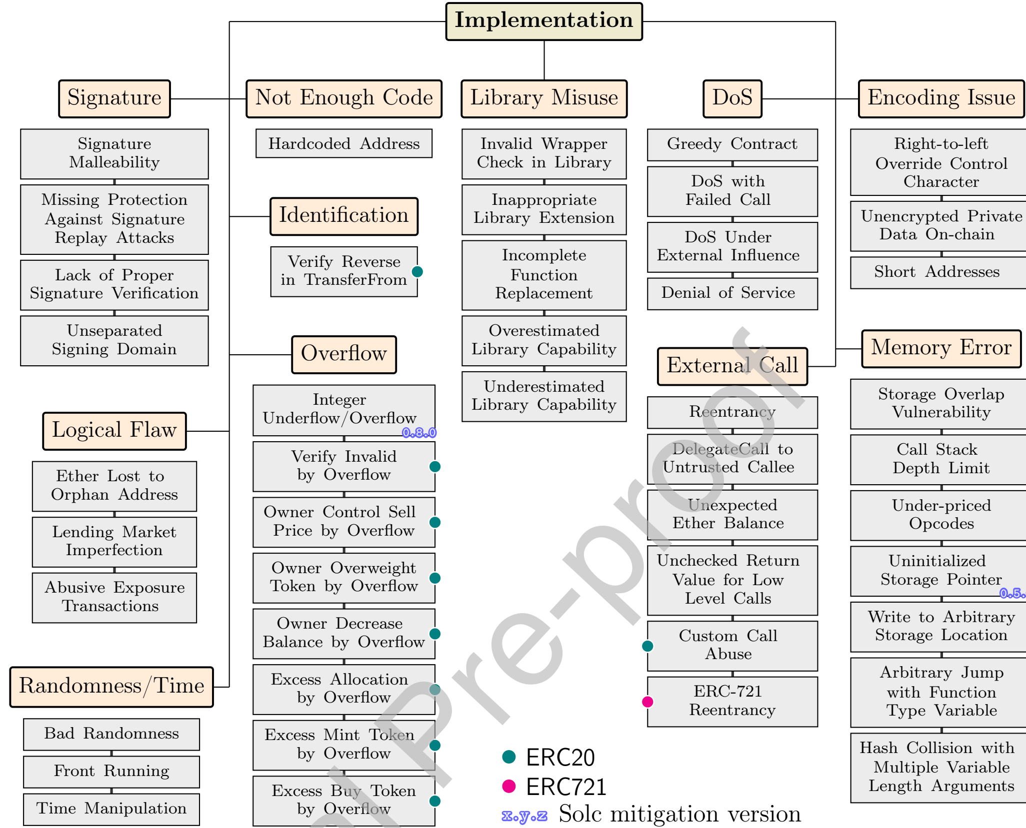


Figure 6: Final classification of vulnerabilities (focus on implementation logic defects).

Application layer, corresponding to the tag Application. We then discriminated code flaws based on their root cause (the why dimension). We have two root causes yielding to code flaws: bugs in the implementation logic, tagged Implementation; and code inconsistencies, tagged Inconsistency. We inserted 23 defects in the first category and 14 defects in the second. Figure 7 depicts all code flaws grouped by root tag.

Considering also the attack vector (the how dimension), we noted that the following exploit tags can be applied to code flaws: 7 defects fall into Gas Related, all belonging to Implementation; 4 defects fall into Not Enough Code, all belonging to Implementation; 2 defects fall into Too Much Code, all belonging to Implementation; 2 defects fall into Library Misuse, all belonging

Journal Pre-proof

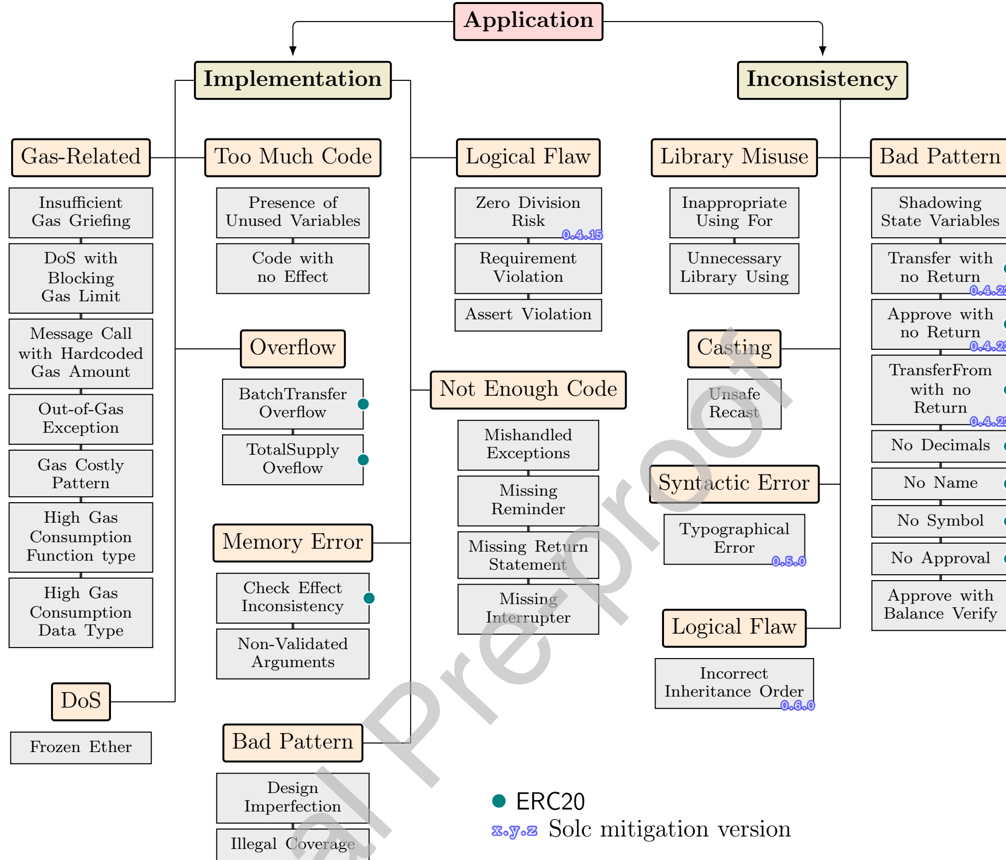


Figure 7: Final classification of code flaws (focus on Application Layer).

to Inconsistency; 1 defect falls into Casting, belonging to Inconsistency; 1 defect falls into DoS, belonging to Implementation; 2 defects falls into Overflow, all belonging to Implementation; 1 defect falls into Syntactic Error, belonging to Inconsistency; 4 defects falls into Logical Flaw, specifically 3 belonging to Implementation and 1 belonging to Inconsistency; and 11 defects falls into Bad Pattern, specifically 2 belonging to Implementation and 9 belonging to Inconsistency.

Orthogonally, we can classify code flaws in terms of topic tags. Indeed, 10 code flaws are related to ERC-20 tokens, tagged ERC20 (7 belonging to Inconsistency and 3 to Implementation); while 6 code flaws have a mitigation version, tagged x.y.z.

Journal Pre-proof

4.2. Examples of Collected Defects

In this last part of the section, we will detail how the leaves of the ontology presented in Figures 4, 5, 6, and 7 look like. Specifically, we will present one vulnerability example and one code flaw example. All the defect information presented below has been incorporated into the ontology, whose implementation is discussed in Section 5.

Example of vulnerability. As an example of vulnerability, we will present a defect listed in the SWC classification: *Delegate Call to Untrusted Callee* (SWC-112). In our ontology, such vulnerability falls under the **Application** category, specifically under the **Implementation** sub-category and the **External Call** sub-sub-category. The vulnerability is also known as *"Delegatecall Injection"*, and there is no version of the Solidity compiler mitigating the defect. A real-world smart contract vulnerable is the `WalletLibrary` contract developed by *Parity Technologies*[4], that can be found on Etherscan at the address `0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4`[5].

Description

To facilitate code reuse, the EVM provides an opcode, `delegatecall`, to insert the bytecode of called contracts into the bytecode of the calling contract. A delegate call is identical to a message call except that the code at the destination address is executed in the context of the calling contract and `msg.sender` and `msg.value` do not change their values. This allows a smart contract to dynamically load code from a different address at runtime (deposit, current address, and balance still refer to the calling contract). Consequently, a called (malicious) contract can directly modify (or manipulate) the caller's state variables. This vulnerability is caused by the fact that a called contract can update the state variables of the caller contract. Calling untrusted contracts is very dangerous, as the code in the destination address can change any of the caller's storage variables and has full control over the caller's balance. The vulnerability can be completely prevented by declaring as a library (which is stateless) a contract that is meant to be accessed via `delegatecall`.

[4] Parity Technologies: `https://www.parity.io`

[5] Link: `https://etherscan.io/address/0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4#code`

Journal Pre-proof

Journal Pre-proof

Buggy Code

Here is an example of a vulnerable Solidity code. The smart contract `HackMe` is vulnerable to attacks exploiting a `delegatecall` to the `Lib` smart contract.

```
contract HackMe {
  address public owner;
  Lib public lib;

  constructor(Lib _lib) {
    owner = msg.sender;
    lib = Lib(_lib);
  }

  fallback() external payable {
    address(lib).delegatecall(msg.data);
  }
}
```

```
contract Lib {
  address public owner;

  function pwn() public {
    owner = msg.sender;
  }
}
```

It is not obvious that the owner of the `HackMe` contract can be changed since there is no function within `HackMe` to do so. However, an attacker can hijack the contract by exploiting the `delegatecall`.

Attack Vector

Here is an example of a malicious Solidity code. The smart contract `Attack` can exploit the vulnerability present in the `HackMe` contract.

```
contract Attack {
  address public hackMe;

  constructor(address _hackMe) {
    hackMe = _hackMe;
  }

  function attack() public {
    hackMe.call(abi.encodewithSignature("pwn()"));
  }
}
```

The attack works as follows. Consider two users: Alice, the victim; and Eve, an attacker.

- Alice distributes `Lib`
- Alice distributes `HackMe` with the address of `Lib`
- Eve distributes `Attack` with the address of `HackMe`
- Eve calls `Attack.attack()`

- Attack is now the owner of HackMe

Let us analyze what happened in detail. When Eve calls Attack.attack(), the HackMe's fallback function is called by sending the pwn() selector function. HackMe forwards the call to Lib using delegatecall. Here msg.data contains the pwn() function selector. This tells the EVM to call the pwn() function inside Lib. The pwn() function updates the owner to msg.sender, and delegatecall executes Lib's code using the context of HackMe. As a result, HackMe's storage is updated with msg.sender, which points to the caller of HackMe, that is, the Attack contract. As a result Eve becomes the owner of the HackMe contract.

Patch Code

Here is an example of a secure version of the HackMe smart contract.

```
contract HackMePatched {
  address public owner;
  address public libAddr;

  constructor(address _libAddr) {
    owner = msg.sender;
    libAddr = _libAddr;
  }

  modifier onlyOwner() {
    require(msg.sender == owner);
    _;
  }

  function setLibAddr(address _libAddr) public onlyOwner {
    libAddr = _libAddr;
  }

  fallback() external payable {
    libAddr.delegatecall(msg.data);
  }
}
```

Now, the fallback function uses delegatecall only on a controlled address pointed by libAddr. In addition, only the owner of HackMePatched can change libAddr.

Example of code flaw. As an example of code flaw, we will present a defect again listed in the SWC classification: *Incorrect Inheritance Order* (SWC-125). In our ontology, such code flaw falls under the Application category, specifically under the Inconsistency sub-category and the Logical Flaw sub-

Journal Pre-proof

sub-category. The code flaw has not other known names, and it has been mitigated in the version `v0.6.0` of the Solidity compiler.

Description

Solidity supports multiple inheritance, which means that a contract can inherit multiple contracts. Multiple inheritance introduces an ambiguity called the *Diamond Problem*: if two or more base contracts define the same function, which one should be called in the child contract? Solidity addresses this ambiguity using inverse C3 linearization, which establishes a priority among the base contracts. In this way, the base contracts have different priorities, so the order of inheritance is fundamental. Neglecting the order of inheritance can lead to an unexpected behavior of the EVM.

Buggy Code

Here is an example of a flawed Solidity code. The smart contract `OwnersCankill` incorrectly inherits from contracts `AdminChecker` and `GuestChecker` according to C3 linearization, yielding a runtime unexpected behavior.

```
pragma solidity 0.5.17;

contract AdminChecker {
  address admin = msg.sender;
  function roleCheck() internal view returns (bool) {
    return msg.sender == admin;
  }
}

contract GuestChecker {
  address guest = msg.sender;
  function roleCheck() internal view returns (bool) {
    return msg.sender == guest;
  }
}

contract OwnersCankill is AdminChecker, GuestChecker {
  function kill() external{
    require(roleCheck(), "Not an Admin");
    selfdestruct(msg.sender);
  }
}
```

Journal Pre-proof

Journal Pre-proof

Replication Scenario

In the example shown above, there are two contracts `AdminChecker` and `GuestChecker` checking if the caller is an administrator or a guest user, respectively. Both contracts have a function `roleCheck()` returning true if the caller has the corresponding role. These checkers are inherited from the `OwnersCanKill` contract. The order of inheritance is important here because it will decide which `roleCheck()` function to call. Guest users should not be able to call the `kill()` function, but according to C3 linearization, we have the following function call path:

1. `OwnerCanKill.kill()` followed by
2. `GuestChecker.roleCheck()` followed by
3. `AdminChecker.roleCheck()`

This will cause unexpected behaviour and allow guest users to call `selfdestruct` on the contract and transfer all funds to their accounts.

Patch Code

This code flaw is harmless from Solidity `v0.6.0`. The compiler will throw an error in case of incorrect inheritance order.

5. Visualization Tool

To make the ontology easily accessible, we first encoded the ontology into a formal language and then implemented a visualization tool to navigate its structure and perform queries on it. In the following, we briefly discuss the tool implementation and showcase some use-case scenarios. A replication package containing the tool is publicly available on Zenodo [35].

5.1. Ontology Implementation

The ontology has been modeled by using the *Resource Description Framework* (RDF) format, a widely adopted standard proposed by W3C[6] for describing and exchanging graph data. The model has been defined with the help of Protégé [36], one of the most common open-source tools used for creating and editing ontologies. The classification structure reported in Section 4 has been encoded into an RDF schema.

[6]World Wide Web Consortium: `https://www.w3.org`

Journal Pre-proof

Journal Pre-proof

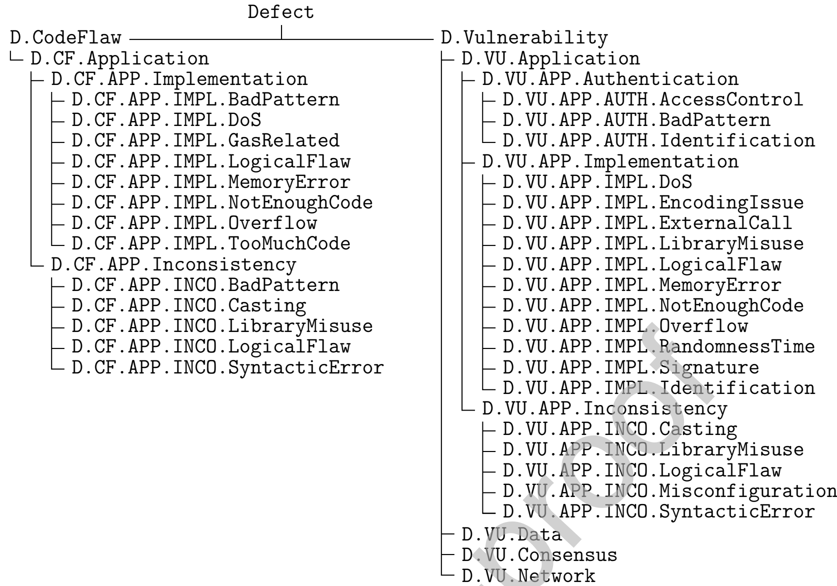


Figure 8: RDF Class hierarchy

Each category presented in the previous section has been implemented as an RDF *Class* in Protégé, with hierarchical information encoded as sub-class relations. Class names reflect the nesting level of a Class in the hierarchy, thus making each class in the ontology unique. For instance, the category Application under the Code Flaw branch of the classification is encoded into a Class named `D.CF.Application`, where the prefix `D.CF` indicates that the Class is a sub-class of `CodeFlaw`, which is, in turn, a sub-class of `Defect` (the root of the hierarchy). Similarly, the Application category under the Vulnerability branch has the name `D.VU.Application`. In the diagram in Figure 8, the full Class hierarchy is reported. Actual defects, either code flaws or vulnerabilities, are then instances of a Class in the hierarchy. We filled in the RDF ontology by instantiating a Class for each of the 135 defects found in the literature review, adhering to the classification reported in Figures 4, 5, 6, and 7.

After Class hierarchy, we defined RDF *Data Properties* in Protégé, that is, the (typed) Class fields corresponding to defect information gathered during the SLR process. Each defect is modeled by using ten Data Properties.

ID of type `integer` is the unique identifier of the defect.

Journal Pre-proof

Name of type `string` is the name assigned to the defect.

Alias of type `list[string]` is the list of alternative names for the defect (if any).

Description of type `string` is the description of the defect.

Code of type `string` is the example code for the defect (if available), including buggy and patched code.

Example of type `string` is the description of the code example related to the defect (if available).

ContractReference of type `string` is the reference to the name/address of a smart contract exposing the defect (if available).

MitigationVersion of type `string` is the version of the Solidity compiler from which the defect was patched (if available).

SWC of type `integer` is the entry of the SWC classification referring to the defect (if available).

Tag of type `string` is the topic tag associated with the defect (if any). Current tags available are ERC20 and ERC721.

After defining Data Properties, the final step consists of the definition of all RDF *Individuals* in Protégé, that is, all instances of the previously described Class and corresponding to the 135 defects in the ontology. We instantiated a Class for each defect and filled in all Data Properties. As name of a Class instance, we have chosen to use its Id and its Name. For instance, the individual representing the vulnerability *Floating Pragma* (SWC-103), instance of the Class `D.VU.APP.INCO.Misconfiguration`, has been given the name `D112.Floating_Pragma`.

During the literature review, it was not possible to obtain all information for all defects (for instance, the mitigation version is not available for all defects). Therefore, not all Individuals have Data Properties completely filled in. Nevertheless, information about Name, Alias, and Description is present for all 135 defects.

Journal Pre-proof

Journal Pre-proof

Extensibility. By adopting a well-known modeling language (RDF), the ontology is easily extensible. New defects can be added by simply creating new individuals and new tags/categories can be added by providing the implementation of new classes. Furthermore, information about currently modeled defects can be easily refined by populating empty Data Properties (for instance by specifying a mitigation version when available).

5.2. Ontology Visualizer

The EDOV visualization tool consists in a Python program that builds a knowledge graph from the ontology RDF file exploiting the `kglab` [37] Python package. The graph is then displayed by creating a dynamic HTML page.

We developed an Angular [38] project to expose the graph. The ontology visualization and exploration are supported by additional navigation functionalities (e.g., filtering and search). The visualizer hence consists in a single page application using Angular and the `PrimeNG` [39] Angular library to manage the graphic components. From the HTML base page created with the Python program, the datasets necessary for the ontology representation are extracted. These are essentially two arrays of objects containing information about the structure's nodes and edges, respectively. At this point, the `vis.js` [40] Javascript library comes into play, which allows the ontology graph to be visualized graphically and provides some useful built-in functionalities (for instance, the "on click" event on defect nodes, or a method to focus on clicked nodes).

Overview exploration. Once the page containing EDOV is open, the tool interface is similar to the one in Figure 9. Then, users can use the mouse to move around the area, dragging and using the mouse wheel to zoom in and zoom out. In this way, users can manually search within the entire ontology for the defect they need to know about.

Defect focus. Once the defect has been identified, the corresponding node can be clicked on. In this case, a window will open on the right-hand side of the page, allowing the user to view detailed information. This is what happens in Figure 10b, which shows a detail of the Floating Pragma vulnerability, one of the two child entities of the `D.VU.APP.INCO.Misconfiguration` category. To help smart contract developers, in the area where the Solidity code is shown, a "copy" button has been added to allow the entire code to be taken and copied to an IDE.

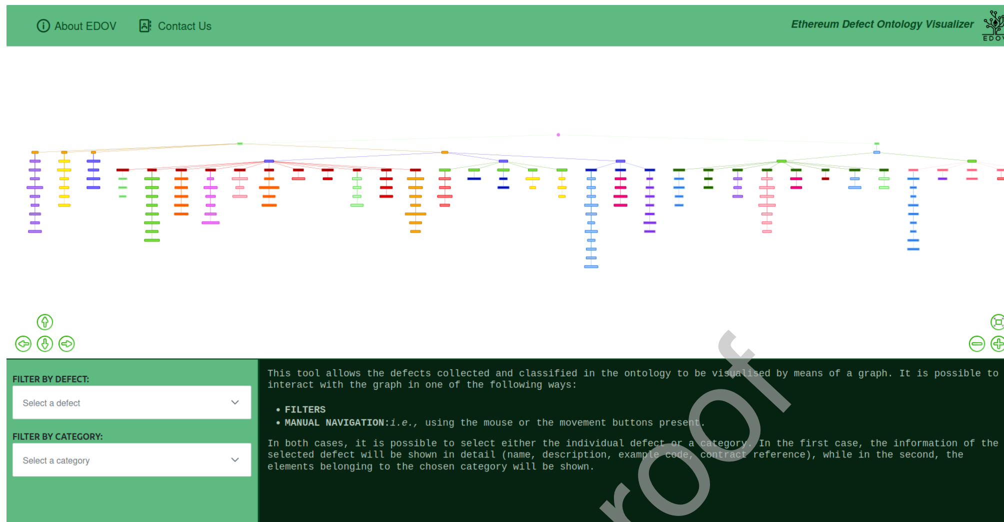


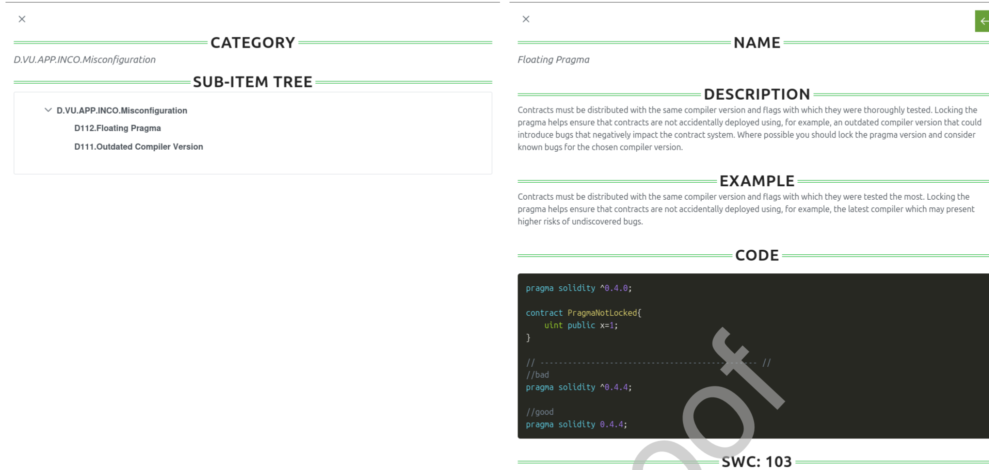
Figure 9: EDOV visualization tool - Initial screen.

Hierarchy exploration. Additionally, it is also possible to click on an internal node in the ontology (i.e., a node corresponding to a Class). In this case, a window containing the textual transcription of the hierarchical representation of the node sub-tree will open on the right-hand side of the page. This is what happens in Figure 10a, where the category clicked is `D.VU.APP.INCO.Misconfiguration`. In this case, there are no other sub-classes but only two vulnerability entities (i.e., two Individuals in the ontology) each clickable to consult its details. This interactive view allows users to expand the various nodes, displaying the children in detail. If one chooses to view the details of a defect from this point, what is shown will be the same as above, except for a button that will allow one to return to the hierarchy view. There are also on-screen buttons for moving around the page. These also include a button to reset the view and return to the starting point.

Filter and search. Since searching for a single defect within the tree can be time-consuming, EDOV provides two filtering functions. These filters are nothing more than two drop-down menus containing the list of all defects and the list of all categories (i.e., internal nodes) respectively. The filters also allow text searching.

Active focus change. For ease of viewing, each time a node is selected, either by filter or manual selection, the focus is shifted to that node. A zoom is

Journal Pre-proof



(a) Category focus. (b) Defect focus.

Figure 10: Ontology graphical interaction examples with EDOV.

then performed on the selected node and the latter slightly changes color to highlight it.

Tool Availability

EDOV is currently an early-stage prototype and has not been publicly released yet. A snapshot of the tool can be found in our replication package, publicly available on Zenodo [35]. We plan to release EDOV soon, and to provide public web-based access to its functionality. The ontology described in Section 4 can also be found at the link above encoded as an RDF file.

6. Conclusion

With this research we aimed to create a comprehensive ontology of known defects in Ethereum and its smart contracts. A systematic literature review identified and categorized known defects, providing real-world examples to illustrate their potential consequences and mitigation strategies. We also made uniform the terminology, proposing a clear definition of security vulnerability and code flaw in the Ethereum context. To facilitate defect understanding, we developed EDOV, a visualization tool that allows users to search, filter,

Journal Pre-proof

Journal Pre-proof

and explore the defect knowledge base distilled. EDOV's hierarchical structure groups similar defects, making it easier to mitigate risks and define enforcing mechanisms.

Our findings revealed a significant lack of existing research on the full spectrum of Ethereum smart contract defects. Many existing studies are limited in scope, focusing on common errors without distinguishing between (security) vulnerabilities and code flaws. Additionally, these studies often lack detailed explanations and code examples, hindering developers' understanding of potential risks.

The results of this research have several implications. The developed ontology provides a valuable foundation for future research on Ethereum smart contract defects by introducing a comprehensive knowledge base focusing specifically on such aspects. By identifying and categorizing common vulnerabilities and code flaws, researchers can develop guidelines and standards to improve smart contract security and dependability. Moreover, this knowledge base can be used to create or enhance code analysis tools for detecting Ethereum defects.

While this research presents promising results, it is important to note that it may be limited by the selection criteria used and the completeness of the information sources consulted. Future work could expand the ontology, enhance EDOV's functionality, and explore defects in other smart contract languages and blockchain platforms. In particular, we plan to integrate the ontology with defects from community-reported incidents (e.g., DeFi Hack Labs) and to enrich the tag system with severity and exploitability metrics (e.g., DASP10 risk levels, SWC severity ratings).

In conclusion, this research contributes to the field of Ethereum smart contract security and dependability by providing a comprehensive resource for understanding and mitigating Ethereum defects. It highlights the importance of ongoing research and development to ensure the integrity and reliability of blockchain-based systems.

Journal Pre-proof

CRediT author statement

Michele Pasqua: Conceptualization, Methodology, Investigation, Supervision, Writing - Original Draft

Sofia Mari: Investigation, Data Curation, Software,

Ferdinando Santoro: Investigation, Data Curation, Software

Mariano Ceccato: Writing - Review & Editing, Supervision

Journal Pre-proof

Declaration of Interest Statement

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

The author is an Editorial Board Member/Editor-in-Chief/Associate Editor/Guest Editor for this journal and was not involved in the editorial review or the decision to publish this article

References

[1] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system (2008). URL http://www.bitcoin.org/bitcoin.pdf

[2] C. Chambers, Forbes - ethereum starts its defi moon shot, [Accessed: 2020-07-28] ([n. d.]). URL https://www.forbes.com/sites/investor/2020/07/23/ethereum-starts-its-defi-moon-shot/#7d34b7ff6ae3

[3] M. Peck, Ethereum's $150-million blockchain-powered fund opens just as researchers call for a halt, [Accessed: 2023-08-12] ([n. d.]). URL https://spectrum.ieee.org/ethereums-150-million-dollar-dao-opens-for-business-

[4] G. Prisco, The DAO raises more than $117 million in world's largest crowdfunding to date, [Accessed: 2020-07-28] ([n. d.]). URL https://bitcoinmagazine.com/articles/the-dao-raises-more-than-million-in-world-s-largest-crowdfunding-to-date-1463422191

[5] K. Li, Y. Wang, Y. Tang, DETER: Denial of ethereum txpool services, in: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 1645—-1667. doi:10.1145/3460120.3485369.

[6] Z. He, Z. Li, A. Qiao, X. Luo, X. Zhang, T. Chen, S. Song, D. Liu, W. Niu, Nurgle: Exacerbating resource consumption in blockchain state storage via MPT manipulation, in: 2024 IEEE Symposium on Security and Privacy (SP), IEEE Computer Society, Los Alamitos, CA, USA, 2024, pp. 2180–2197. doi:10.1109/SP54263.2024.00125.

Journal Pre-proof

[7] N. Group, Decentralised application security project (dasp) top 10, [Accessed: 2023-08-12] ([n. d.]).
URL https://dasp.co

[8] SmartContractSecurity, Smart contract weakness classification (swc), [Accessed: 2023-08-12] ([n. d.]).
URL https://swcregistry.io

[9] B. Vitalik, Ethereum: A next generation smart contract & decentralized application platform (2014).
URL https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf

[10] C. Dannen, Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners, 1st Edition, Apress, USA, 2017.

[11] A. Antonopoulos, G. Wood, G. Wood, Mastering Ethereum: Building Smart Contracts and DApps, O'Reilly Media, Incorporated, 2018.
URL https://books.google.it/books?id=SedSMQAACAAJ

[12] A. M. Antonopoulos, Mastering Bitcoin: Programming the Open Blockchain, 2nd Edition, O'Reilly Media, Inc., 2017.

[13] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, M. Imran, An overview on smart contracts: Challenges, advances and platforms, Future Generation Computer Systems 105 (2020) 475–491. doi:10.1016/j.future.2019.12.019.

[14] ethereum.org, Ethereum development documentation, [Accessed: 2023-08-12] ([n. d.]).
URL https://ethereum.org/en/developers/docs

[15] soliditylang.org, Solidity documentation, [Accessed: 2023-08-12] ([n. d.]).
URL https://solidity.readthedocs.io

[16] Etherscan, The ethereum blockchain explorer, [Accessed: 2021-10-04] ([n. d.]).
URL https://etherscan.io/

Journal Pre-proof

Journal Pre-proof

[17] ethervm.io, Ethervm, [Accessed: 2023-08-12].
URL https://ethervm.io

[18] Ethereum, Solidity documentation - creating contracts, [accessed: 2020-12-29] ([n. d.]).
URL https://docs.soliditylang.org/en/v0.8.0/contracts.html#creating-contracts

[19] G. Wood, Ethereum: A secure decentralised generalised transaction ledger.

[20] H. Chen, M. Pendleton, L. Njilla, S. Xu, A survey on ethereum systems security: Vulnerabilities, attacks, and defenses, ACM Comput. Surv. 53 (3) (jun 2020). doi:10.1145/3391195.

[21] K. Wu, Y. Ma, G. Huang, X. Liu, A first look at blockchain-based decentralized applications, Software: Practice and Experience 51 (10) (2021) 2033–2050. doi:https://doi.org/10.1002/spe.2751.

[22] F. Schär, Decentralized Finance: On Blockchain- and Smart Contract-Based Financial Markets, Review 103 (2) (2021) 153–174. doi:10.20955/r.103.153-74.

[23] Bloomberg.com, Rise of crypto market's quiet giants has big market implications, [Accessed: 2023-08-12] ([n. d.]).
URL https://www.bloomberg.com/news/articles/2021-03-19/rise-of-crypto-market-s-quiet-giants-has-big-market-implications

[24] Ethereum, Ethereum improvement proposals, [accessed: 2020-12-29] ([n. d.]).
URL https://eips.ethereum.org

[25] A. Ghaleb, K. Pattabiraman, How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020, Association for Computing Machinery, New York, NY, USA, 2020, pp. 415–427. doi:10.1145/3395363.3397385.

Journal Pre-proof

[26] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, T. Chen, Defining smart contract defects on ethereum, IEEE Trans. Softw. Eng. 48 (1) (2022) 327–345. doi:10.1109/TSE.2020.2989002.

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

[28] L. Wenkai, B. Jiuyang, L. Xiaoqi, P. Hongli, N. Yuanzheng, Z. Yuqing, A survey of defi security: Challenges and opportunities, Journal of King Saud University - Computer and Information Sciences 34 (10, Part B) (2022) 10378–10404. doi:https://doi.org/10.1016/j.jksuci.2022.10.028.

[29] G. Arunima, G. Shashank, D. Amit, K. Neeraj, Security of cryptocurrencies in blockchain technology: State-of-art, challenges and future prospects, Journal of Network and Computer Applications 163 (2020) 102635. doi:https://doi.org/10.1016/j.jnca.2020.102635.

[30] C. Hanting, Z. Pengcheng, D. Hai, X. Yan, J. Shunhui, L. Wenrui, A survey on smart contract vulnerabilities: Data sources, detection and repair, Information and Software Technology 159 (2023) 107221. doi:https://doi.org/10.1016/j.infsof.2023.107221.

[31] B. A. Kitchenham, S. Charters, Guidelines for performing systematic literature reviews in software engineering, Tech. Rep. EBSE 2007-001, Keele University and Durham University Joint Report (7 2007).
URL https://www.elsevier.com/__data/promis_misc/525444systematicreviewsguide.pdf

[32] Scopus, Scimago journal rank 2023, [Accessed: 2023-08-12] ([n. d.]).
URL https://www.scimagojr.com/journalrank.php

[33] ICORE Conference Portal, CORE 2023 conference ranking, [Accessed: 2023-08-12] ([n. d.]).
URL https://portal.core.edu.au/conf-ranks/

[34] T. Zimmermann, Card-sorting: From text to themes, in: T. Menzies, L. Williams, T. Zimmermann (Eds.), Perspectives on Data Science for Software Engineering, Morgan Kaufmann, Boston, 2016, pp. 137–141. doi:10.1016/B978-0-12-804206-9.00027-1.

Journal Pre-proof

URL https://www.sciencedirect.com/science/article/pii/B9780128042069000271

[35] M. Pasqua, S. Mari, F. Santoro, M. Ceccato, An ontology of defects for ethereum and its smart contracts (replication package) (2025).
URL https://doi.org/10.5281/zenodo.17201280

[36] S. University, Proégé, [Accessed: 2023-08-12] ([n. d.]).
URL http://protege.stanford.edu/

[37] DerwenAI, kglab, [Accessed: 2023-08-12] ([n. d.]).
URL https://github.com/DerwenAI/kglab/

[38] Google, Angular, [Accessed: 2023-08-12] ([n. d.]).
URL https://angular.dev/

[39] primefaces, Primeng, [Accessed: 2023-08-12] ([n. d.]).
URL https://github.com/primefaces/primeng/

[40] vis.js Community, vis.js, [Accessed: 2023-08-12] ([n. d.]).
URL https://visjs.org/

[41] smartbugs, Smartbugs-curated (dataset), [Accessed: 2020-07-28] ([n. d.]).
URL https://github.com/smartbugs/smartbugs-curated

[42] gsalzer, Cgt (dataset), [Accessed: 2020-07-28] ([n. d.]).
URL https://github.com/gsalzer/cgt

[43] smartbugs, Smartbugs-wild (dataset), [Accessed: 2020-07-28] ([n. d.]).
URL https://github.com/smartbugs/smartbugs-wild

[44] Franklinliu, Spcon artifact (dataset), [Accessed: 2020-07-28] ([n. d.]).
URL https://github.com/Franklinliu/SpCon-Artifact

[45] sujeetc, Scrawid (dataset), [Accessed: 2020-07-28] ([n. d.]).
URL https://github.com/sujeetc/ScrawlD

[46] renardbebe, Smart-contract-benchmark-suites (dataset), [Accessed: 2020-07-28] ([n. d.]).
URL https://github.com/renardbebe/Smart-Contract-Benchmark-Suites

Journal Pre-proof

Journal Pre-proof

[47] InPlusLab, Reentrancystudy-data (dataset), [Accessed: 2020-07-28] ([n. d.]).
URL https://github.com/InPlusLab/ReentrancyStudy-Data

Journal Pre-proof