Empirical Software Engineering (2024) 29:101
https://doi.org/10.1007/s10664-024-10446-8

OpenSCV: an open hierarchical taxonomy for smart contract vulnerabilities

Fernando Richter Vidal[1] · **Naghmeh Ivaki**[1] · **Nuno Laranjeiro**[1]

Accepted: 8 January 2024 / Published online: 18 June 2024

Abstract

Smart contracts are nowadays at the core of most blockchain systems. Like all computer programs, smart contracts are subject to the presence of residual faults, including severe security vulnerabilities. However, the key distinction lies in how these vulnerabilities are addressed. In smart contracts, when a vulnerability is identified, the affected contract must be terminated within the blockchain, as due to the immutable nature of blockchains, it is impossible to patch a contract once deployed. In this context, research efforts have been focused on proactively preventing the deployment of smart contracts containing vulnerabilities, mainly through the development of vulnerability detection tools. Along with these efforts, several heterogeneous vulnerability classification schemes appeared (e.g., most notably DASP and SWC). At the time of writing, these are mostly outdated initiatives, even though new smart contract vulnerabilities are consistently uncovered. In this paper, we propose OpenSCV, a new and Open hierarchical taxonomy for Smart Contract vulnerabilities, which is open to community contributions and matches the current state of the practice while being prepared to handle future modifications and evolution. The taxonomy was built based on the analysis of the existing research on vulnerability classification, community-maintained classification schemes, and research on smart contract vulnerability detection. We show how OpenSCV covers the announced detection ability of the current vulnerability detection tools and highlight its usefulness in smart contract vulnerability research. To validate OpenSCV, we performed an expert-based analysis wherein we invited multiple experts engaged in smart contract security research to participate in a questionnaire. The feedback from these experts indicated that the categories in OpenSCV are representative, clear, easily understandable, comprehensive, and highly useful. Regarding the vulnerabilities, the experts confirmed that they are easily understandable.

Communicated by: Xin Xia

✉ Fernando Richter Vidal
fernandovidal@dei.uc.pt

Naghmeh Ivaki
naghmeh@dei.uc.pt

Nuno Laranjeiro
cnl@dei.uc.pt

[1] University of Coimbra, Centre for Informatics and Systems of the University of Coimbra, Department of Informatics Engineering, Coimbra, Portugal

Keywords Blockchain · Smart contracts · Vulnerabilities · Classification · Taxonomy

1 Introduction

Smart contracts play an important role in advancing blockchain as they expand the application of the technology to various domains (e.g., finance (Hewa et al. 2021), education (Grech and Camilleri 2017), healthcare (Agbo et al. 2019), government (Geneiatakis et al. 2020)). While they are essential for the consolidation and expansion of the technology, they also bring serious risks, namely those associated with the potential presence of vulnerabilities that can affect the security of the blockchain system (Atzei et al. 2017).

Just as conventional programs, smart contracts are being deployed with residual software faults (i.e., bugs or defects), including security vulnerabilities (i.e., internal faults that enable external events to harm the system) (Qian et al. 2022; Avizienis et al. 2004). However, the consequences of deploying a faulty contract have particular characteristics in the context of blockchain systems, such as: i) if faulty code is identified, the respective contract cannot be patched, it must be terminated, and a new one should be created (Zou et al. 2019); ii) once the potentially erroneous data (generated/updated by faulty contracts) has been stored in the blockchain, there is no way to change it, i.e., to undo the respective transactions (and subsequent transactions that rely on this data) (Yaga et al. 2018); and iii) if the faulty contract has been executed, the associated impact may be irreparable (e.g., reputation costs) (Antonopoulos and Wood 2018).

Several initiatives have been created that ultimately aim at contributing to the development of more secure smart contracts. Among these initiatives, we find three main types: i) New smart contract programming languages (e.g., Clarify (Blockstack 2021), Vyper (Kaleem et al. 2020), Obsidian (Coblenz 2019)), which aim at increasing protection against vulnerabilities; ii) New vulnerability detection tools (e.g., Mythril (ConsenSys 2021), Neucheck (Lu et al. 2019), Bose et al. (2022), SoliDetector (Hu et al. 2023)), which have the main goal of detecting vulnerabilities in smart contracts so that vulnerable contracts do not reach the deployment phase; and also, iii) vulnerability classifications that mostly allow knowledge regarding vulnerabilities to be identified in a standard manner and systematized.

The existence of vulnerability (or software defects, in general) classifications is quite important, as we can observe by the research and industry effort associated with well-known cases like OWASP (OWASP Foundation 2001), NVD (government 1999), CVE (MITRE Corporation 1999), CWE (CWE Community 2009), or, in the case of smart contracts most notably by SWC (SmartContractSecurity 2020), and DASP (NCCGroup 2021). Generally, they raise the level of awareness among developers and may allow, in a uniform manner, for development tools to assist developers regarding defects being placed in the code. They may also help in the design and development of vulnerability detection tools and in the assessment of their detection capabilities (Durieux et al. 2020; Hu et al. 2021; di Angelo and Salzer 2019). This case also holds for programming languages. It is known that languages, such as Obsidian, have benefited from the systematized knowledge of vulnerabilities. There are even studies that use taxonomies as a basis for comparing different programming languages with respect to the protection offered against certain types of vulnerabilities (Kaleem et al. 2020).

At the time of writing, vulnerability classifications for smart contracts have significant limitations. First, these classifications are often **outdated**, such as well-known schemes like DASP or SWC that have not been updated since 2018. This largely differs from the state of the practice, in which we find cases of tools like Securify2 (Tsankov 2018) already detecting

several vulnerabilities for which there is no accurate description in the classifications. As in other software areas, with new vulnerabilities being continuously discovered, having a flexible way of integrating them (and possibly restructuring the classification) is crucial.

Second, vulnerability naming and classification schemes are being defined using **arbitrary nomenclatures**. This is easily visible just by analyzing a few of the most cited papers in vulnerability detection, e.g., (Luu et al. 2016; Tsankov et al. 2018; Kalra et al. 2018). The lack of a standard nomenclature leads to verification tools mostly using arbitrary names to present their result. For instance, SmartCheck (Tikhomirov et al. 2018) and Slither (Feist et al. 2019) respectively use *balance equality* and *incorrect-equality/locked-ether* to refer to the same vulnerability. As a result, it is very difficult to compare the effectiveness of different tools. As classifications are often built based on different sources, such as different industry tools and several research papers (Rameder et al. 2022), the terms easily end up being inconsistent. This is aggravated when no active maintenance exists, even for known issues. Indeed, **reduced community contribution** is known to be a problem, with the main classifications that are community-oriented (i.e., DASP, SWC) showing residual community activity, many times related to minor problems (e.g., broken links) (NCC Group 2019; SmartContractSecurity 2020).

Third, many times, vulnerability classification schemes mix the characteristics of a certain vulnerability with the effect of exploiting it, how it is exploited, or its impact. This **concept inconsistency** is quite visible in the existing taxonomies. As an example, in Kaleem et al. (2020), *DoS with unbounded operation* is presented as a vulnerability, but it is not possible to understand what exactly the vulnerability is with this name (e.g., it can be a problem in a loop, it can be a malicious call that is externally triggered several times). Instead, the given name refers to the possible impact of exploiting a vulnerability, which should be a separate dimension for characterizing the vulnerability. Similarly, this occurs in DASP (NCC Group 2019), in which one of the categories is precisely *Denial of Service*. Another aspect this latter example shows is that taxonomies are being built with **inadequate granularity**, often too coarse to be really helpful. For instance, the *Denial of Service* category in DASP may refer to *gas limit reached*, *unexpected throw*, *unexpected kill*, or *access control breached*. Moreover, the description is sometimes so short that it may become ambiguous (e.g., *access control breached* may refer to a vulnerability that would simply fit in *access control*, which is another DASP category).

In this paper, we propose OpenSCV, a new hierarchical and Open taxonomy for Smart Contract Vulnerabilities (available at https://openscv.dei.uc.pt), which is open to community's contributions (Vidal et al. 2024b). OpenSCV aims to match the current state of the practice and is prepared to handle future modifications and evolution. To build the taxonomy, we analyzed current smart contract vulnerability classifications and discussed their gaps and limitations. We then analyzed the detection capabilities of 77 smart contract vulnerability detection tools, which resulted in collecting a heterogeneous set of 481 vulnerabilities. We then mapped the vulnerabilities in existing classifications, namely DASP (NCC Group 2019), SWC (SmartContractSecurity 2020), Rameder et al. (2022), and CWE (CWE Community 2009) and further characterized them using the Orthogonal Defect Classification (ODC) (IBM 2013b, a) and a code excerpt. Names were then consolidated and grouped in a structure that was built bottom-up. This process involved two experienced researchers and one early-stage researcher, who revised the proposed taxonomy iteratively in terms of structure, correctness, and uniformity.

We structured OpenSCV in a way that is flexible to changes and evolution by preparing a supporting infrastructure at GitHub. We are able to receive change requests easily and integrate information from new research on vulnerability detection into the taxonomy. All

OpenSCV entries are supported by a code example, with the goal of mitigating possible ambiguities in the description of each vulnerability. We also prepared an initial dataset holding vulnerable contracts (one per each of the vulnerabilities present in OpenSCV) and their respective correction. OpenSCV is available online at https://openscv.dei.uc.pt (Vidal et al. 2024c). The GitHub repository is available at Vidal et al. (2024b) and linked to Zenodo, which permanently hosts the dataset (Vidal et al. 2024a). It is worthwhile mentioning that the taxonomy considers mostly software vulnerabilities and a few software defects considered in the literature to be associated with high-security risks. For simplicity, *we use the term vulnerability throughout the paper* to refer to both cases.

To validate our taxonomy, we conducted an expert-based analysis. We invited 150 experts who are actively involved in smart-contract security research to participate in a questionnaire. We have received 28 responses and additional comments. We then aggregated all responses to calculate an agreement score. Feedback on the vulnerability categories indicated that they are representative, clear, easily understandable, comprehensive, and highly useful, achieving an overall score of 0.78 out of 1.00. Regarding the vulnerabilities, the feedback highlighted their ease of understanding, resulting in an overall score of 0.76 out of 1.00.

The rest of this paper is organized as follows. Section 2 discusses the related work and limitations of current vulnerability classification schemes. Section 3 presents the process followed to build the taxonomy and overviews the final outcome. Section 4 presents the taxonomy structure and provides a brief description of all vulnerabilities included in the taxonomy. Section 5 characterizes and discusses the coverage of the taxonomy in perspective with the state-of-the-art and presents the validation results. Section 6 presents the threats to the validity of this work, and finally, Section 7 concludes this paper.

2 State of the Art

This section presents the quality properties of taxonomies and then discusses the existing classification schemes for smart contract vulnerabilities. The classification schemes presented have their origins in a) existing research on smart contract vulnerability classification, b) community-oriented initiatives, and c) vulnerability detection research. The section closes with a discussion of the gaps and limitations of the existing classifications.

2.1 Taxonomy Quality Properties

We analyzed a set of reference works centered around the definition of taxonomies as well as critical analyses of vulnerability taxonomies (Bishop and Bailey 1996; Lindqvist and Jonsson 1997; Mann and Christey 1999; Rameder et al. 2022; Lough 2001; Hansman and Hunt 2005) to identify a set of quality properties criteria, which should be followed when designing a taxonomy that is expected to be long-lived. The following paragraphs discuss the identified properties.

A classification system may benefit from a **hierarchical organization** as it allows to show similar characteristics of related vulnerabilities, which may also be helpful for vulnerability prevention (Bishop and Bailey 1996). A hierarchical structure may be a tree in which each node refers to a category of vulnerabilities, and each leaf corresponds to individual vulnerabilities. Thus, the granularity of the categories should generally vary from large to fine as we traverse the tree from the root to the leaves.

Nodes at a certain tree level must be as uniform as possible, i.e., ideally representing the same **level of abstraction** or a group of vulnerabilities viewed from the same perspective. Obviously, this is quite difficult to achieve because, many times, this has to be balanced with the creation of taxonomy trees that become too complex, which, in the end, may make it less comprehensible or less helpful. Also, sometimes the nature of the problem is simply an unbalanced or heterogeneous (in structure) one, which basically disallows these criteria. Anyway, a uniform taxonomy may contribute to fewer errors (in its use) and, as such, a higher probability of adoption by practitioners. In practice, it may contribute to a taxonomy that is **useful** and **understandable** (Lindqvist and Jonsson 1997) (i.e., understandable by security experts but also by less specialized people).

The selection of names to be used in a classification scheme is particularly important. The name that describes a certain vulnerability must be a **unique identifier**) and **non-ambiguous** (Howard 1997; Lindqvist and Jonsson 1997), meaning that the name and also the associated description must allow not only for easy identification but should include enough information to distinguish it from other vulnerabilities (Mann and Christey 1999; Bishop and Bailey 1996). Whenever possible, existing **terminology** should be used (Lindqvist and Jonsson 1997). The name and characteristics of a certain defect should characterize what the issue is and not additional dimensions, such as the effect of exploiting it. While it is acceptable to understand the effect of exploiting a certain vulnerability starting from its description, the characteristics of the problem itself cannot be omitted and should be clearly identified (Mann and Christey 1999; Bishop and Bailey 1996).

Regardless of the perspective of the individuals using the taxonomy, a certain vulnerability should be classified in the same manner by different individuals (e.g., developers, users, testers). This means that not only the names and structure should be as clear as possible, but also that the process of classifying a certain defect must be made clear (whenever the structure and nomenclature are not sufficient), i.e., there must be a **deterministic** (Krsul 1998) way of classifying a certain defect, which fosters **repeatability** (Howard 1997; Krsul 1998) of using the classification.

Finally, a taxonomy should also allow for **completeness** (Amoroso 1994), i.e., the taxonomy should provide a **good coverage** (Rameder et al. 2022) of the vulnerabilities identified in state of the art or reported by vulnerability detection tools. Also, it should be **open to the community** (i.e., accept new entries from the community) and shareable (i.e., no distribution restrictions) (Mann and Christey 1999). The fact that it is open is also a factor that can contribute to it being **accepted** (Amoroso 1994; Howard 1997).

2.2 Smart Contract Vulnerability Classification Schemes

To the best of our knowledge, the first initiative to classify smart contract vulnerabilities (for Ethereum systems) is proposed in Atzei et al. (2017). The authors listed 12 vulnerabilities, which we listed in Table 1, and implemented nine of the corresponding attacks.

This initial effort is quite relevant but holds some limitations. Some of the selected names do not really specify the nature of the vulnerabilities or are not clear about the problem being characterized (e.g., *call to the unknown*). This limitation was mitigated in Zhou et al. (2022a) and Argañaraz et al. (2020), where the authors tried to make the names used more specific. In Atzei et al. (2017), three categories of issues are proposed: i) Solidity issues (i.e., language weaknesses), ii) EVM issues (i.e., residuals faults in byte code), and iii) Blockchain issues (i.e., vulnerabilities from blockchain technology). Despite allowing an

Table 1 Classification proposed in Atzei et al. (2017)

Level	Vulnerability
Solidity	Call to the unknown
	Gasless send
	Exception disorders
	Type casts
	Reentrancy
	Keeping secrets
EVM	Immutable bugs
	Ether lost in transfer
	Stack size limit
Blockchain	Unpredicable state
	Generating randomness
	Time constraints

initial separation of the vulnerabilities (which may help developers in dealing with them), this scheme does not benefit from the presence of a more complex hierarchy, which is a better fit for cases where we find several interrelated families of vulnerabilities. We have also identified that these three categories may generate some ambiguity as some cases could potentially fit into multiple categories. For example, *Immutable Bugs* could be classified into EVM or Blockchain. Despite this, the separation between the cases referring to the programs (i.e., solidity source code or EVM binary code) and the blockchain platform is helpful. This classification is not available in a public repository, and, due to its age, its coverage is relatively low, accounting for 12 vulnerabilities.

Table 2 overviews the vulnerability classification presented in Kaleem et al. (2020), which has the goal of allowing comparison between the security of the Solidity and Vype languages. The work presents 18 vulnerabilities, along with a detailed explanation for each one, and individual code examples for each vulnerability. Being mostly a list of vulnerabilities, there are no benefits associated with hierarchical structures. Moreover, there is no open public repository associated with the proposal, and the 18 vulnerabilities are nowadays a small amount.

A classification is presented in Argañaraz et al. (2020) with the goal of exposing threats and, ultimately, minimizing the presence of software faults in smart contracts. Table 3 presents the proposed classification in which we find that the faults are separated into two categories: i) security vulnerabilities (i.e., defects that may be exploited by attacks; and ii) functional faults (i.e., faults that violate the program's functionality). Each fault is also associated with a criticality level, which may be useful for getting developers' attention while coding. We found that certain cases, such as *Non-verified maths* and *Malicious libraries*, may refer to the same vulnerability, indicating a potential need for further clarification and refinement of the classification process to address any ambiguity. Similar to the previously presented works, there is no hierarchical structure besides the two groups of faults. Moreover, some names used in the classification are quite specific (i.e., *Use of tx.origin*), which makes it difficult to understand the problem in a more abstract manner. Only 13 faults are considered, with no possibility of expansion. Still, the idea of classifying the faults into two broad concepts of security and functionality is a vision that may be interesting for newer classifications (e.g., targeting different types of systems).

Table 2 Classification proposal in Kaleem et al. (2020)

Vulnerabilities
Integer overflow and underflow
DoS with unbounded operation
Unchecked call return value
Reentrancy
Delegate call injection
Forced Ether to contract
DoS with unexpected revert
Erroneous visibility
Uninitialized storage pointer
Upgradeable contract
Type casts
Insufficient signature information
Frozen Ether
Authentication through tx. Origin
Unprotected suicide
Leaking Ether to arbitrary address
Secrecy failure
Outdated compiler version

A smart contract vulnerability classification is presented in Zhou et al. (2022a), based on a previous work presented in Atzei et al. (2017). In this classification, summarized in Table 4, the groups were maintained (i.e., Solidity, EVM, and Blockchain), but the vulnerability entries were modified (i.e., some names were removed, like *stack size limit* and *gasless send* and other names were included, like *tx. origin* and *default visibility*). The authors linked the proposed names to an external taxonomy, namely CWE (CWE Community 2009), which is helpful for

Table 3 Classification proposal in Argañaraz et al. (2020)

Level	Vulnerability	Impact
Security	Equality on the balance	Average
	Non-verified external call	High
	Use of send instead of transfer	Average
	Denial of a service because of an external contract	High
	Re-entrancy	High
	Malicious libraries	Low
	Use of tx.origin	Average
	Transfer of all the gas	High
Functional	Integer division	Low
	Blocked money	Average
	Non-verified maths	Low
	Dependence on the timestamp	Average
	Unsecure inference	Average

Table 4 Vulnerability classification in Zhou et al. (2022a)

Level	Vulnerability	CWE	Real-word attack
Solidity	Re-entrancy	CWE-841	The DAO Attack
	Arithmetic issues	CWE-682	PoWHcoin attack
	Delegatecall to insecure contracts	CWE-829	Parity Wallet (Second Hack)
	Selfdestruct	CWE-284	Parity Library bug
	Tx.origin	CWE-477	–
	Mishandled exception	CWE-252	King of The Ether attack
	Default visibility	CWE-710	Parity Wallet (First Hack)
	External contract referencing	CWE-829	Honey Pot
EVM	Short address/parameter issues	CWE-88	–
	Freezing Ether	CWE-17	–
Blockchain	Transaction order dependence	CWE-362	Attack on Bancor
	Generating randomness	CWE-330	PRNG contract
	Timestamp dependence	CWE-829	GovernMental attack

understanding each vulnerability, verifying the correctness of the proposed classification, and also for standardization purposes. The proposed classification defines a basic separation of vulnerabilities, mostly distinguishing cases related to the programs from cases related to the platform. Again, the number of vulnerabilities listed is quite small (i.e.,13 vulnerabilities), and the work could benefit from a repository open to community contributions.

Table 5 presents a vulnerability classification proposed by Amiet (2021). The classification is based on two categories: i) core blockchain vulnerabilities (i.e., vulnerabilities related to the blockchain platform) and ii) smart contracts vulnerabilities (i.e., vulnerabilities related to the programs deployed in the blockchain). At the blockchain level, examples are provided (e.g., attacks on the consensus mechanism), whereas, at the contract level, pseudo-code is presented, which clarifies the security issues identified. These two broad groups are a basis for applying the classification to other types of systems. There are no further hierarchical

Table 5 Classification proposed in Amiet (2021)

Group	Vulnerabilities
Core blockchain	Consensus mechanism manipulation
	Underlying cryptosystem vulnerabilities
	Improper blockchain magic validation
	Improper transaction nonce validation
	Denial of service
	Public-key and address mismatch
Smart contract	Reentrancy
	Arithmetic issues
	Unprotected selfdestruct
	Visibility issues
	Weak randomness
	Transaction order dependence

levels present in this taxonomy, and we found vulnerability names that are unclear, such as *Improper Blockchain Magic Validation*, which does not really characterize the technical details involving the vulnerability. As with previous cases, the 12 vulnerabilities represent a quite small number of currently known vulnerabilities.

A classification of 28 vulnerabilities is proposed in Staderini et al. (2020) and was further evolved to categorize a total of 33 vulnerabilities in Staderini et al. (2022). Table 6 presents an overview of the authors' classification, identifying the acronym and name of the vulnerabilities and an associated CWE (CWE Community 2009).

Table 6 Classification proposed in Staderini et al. (2022)

Acr.	*Vulnerability name*	*CWE-ID*
ELT	Ether lost in transfer	
RV	Requirement violation	CWE-20
SA	Short addresses	
Atx	Authorization through tx. origin	
UEW	Unprotected ether withdrawal	
Usd	Unprotected selfdestruct	CWE-284
UWSL	Unprotected write to storage location	
VEF	Visibility of exposed functions	
GR	Generating randomness	CWE-330
MPRA	Missing protection against signature	
SM	Signature malleability	
Ty	Type casts	CWE-345
CPL	DoS costly patterns and loops	
Gs	Gasless send	CWE-400
BU	Blockhash usage	
ML	Malicious libraries	
SF	Secrecy failure	
TD	Timestamp dependency	CWE-668
CU	Call to the unknown	
DUC	Delegatecall to the untrusted callee	
EC	DoS by external contracts	CWE-669
AP	Arithmetic precision order	
IOU	Integer overflow or underflow	CWE-682
AJ	Arbitrary Jump	
FE	Freezing ether	
IGG	Insufficient gas griefing	
Re	Reentrancy	
RLO	Right left override	
TOD	Transaction ordering dependence	
UEB	Unexpected ether balance	CWE-691
ED	Exception disorder	
Us	Unchecked send	
UV	Unchecked call return values	CWE-703

Table 7 Vulnerability classification in Rameder et al. (2022)

Group	Code	Vulnerability
Malicious environment, Transactions or input	1A	Reentrancy
	1B	Call to the unknown
	1C	Exact balance dependency
	1D	Improper data validation
	1E	Vulnerable DELEGATECALL
Blockchain/Environment dependency	2A	Timestamp dependency
	2B	Transaction-ordering dependency (TOD)
	2C	Bad random number generation
	2D	Leakage of confidential information
	2E	Unpredictable state (dynamic libraries)
	2F	Blockhash dependency
Exception & Error handling disorders	3A	Unchecked low level call/send return values
	3B	Unexpected throw or revert
	3C	Mishandled out-of-gas exception
	3D	Assert, require or revert violation
Denial of service	4A	Frozen Ether
	4B	Ether lost in transfer
	4C	DoS with block gas limit reached
	4D	DoS by exception inside loop
	4E	Insufficient gas griefing
Resource consumption & Gas issues	5A	Gas costly loops
	5B	Gas costly pattern
	5C	High gas consumption of variable data type or declaration
	5D	High gas consumption function type
	5E	Under-priced opcodes
Authentication & Access control vulnerabilities	6A	Authorization via transaction origin
	6B	Unauthorized accessibility due to wrong function or state variable visibility
	6C	Unprotected self-destruction
	6D	Unauthorized Ether withdrawal
	6E	Signature based vulnerabilities
Arithmetic bugs	7A	Integer over- or underflow
	7B	Integer division
	7C	Integer bugs or arithmetic issues

Table 7 continued

Group	Code	Vulnerability
Bad Coding and Language Specifics	8A	Type cast
	8B	Coding error
	8C	Bad coding pattern
	8D	Deprecated source language features
	8E	Write to arbitrary storage location
	8F	Use of assembly
	8G	Incorrect inheritance order
	8H	Variable shadowing
	8I	Misleading source code
	8J	Missing logic, logical errors or dead code
	8K	Insecure contract upgrading
	8L	Inadequate or incorrect logging or documentation
Environment Configuration Issues	9A	Short address
	9B	Outdated compiler version
	9C	Floating or no pragma
	9D	Token API violation
	9E	Ethereum update incompatibility
	9F	Configuration error
Eliminated/Deprecated Vulnerabilities	10A	Callstack depth limit
	10B	Uninitialized storage pointer
	10C	Erroneous constructor name

As we can see in Table 6, the additional characterization by CWE is quite helpful, although it is not accompanied by a blockchain-specific classification scheme, such as SWC (SmartContractSecurity 2020), which could help in unifying knowledge. The source of information is based on a set of references in Staderini et al. (2020) and Staderini et al. (2022), which do not directly map with the state of the practice (e.g., tools for vulnerability detection). Still, the process for building the classification is insightful and helpful as a way to solidify our own classification, e.g., by allowing verification of our own mapping to CWE.

A consolidated taxonomy is presented in Rameder et al. (2022). The authors were able to collect 54 vulnerabilities reported from different verification tools and grouped them into 10 categories. Table 7 overviews the taxonomy created by the authors.

This classification is more fine-grained than the previously discussed ones. However, there are a few issues with some names given to the vulnerabilities. For instance, it is not obvious to what extent *integer bugs or arithmetic issues - 7C* is different from *Integer over- or underflow - 7A* or *Integer division - 7B*, and there are names like *Gas costly loops* and *Gas costly pattern* which seem very similar. Also, names like *configuration error* are quite generic and could lead to a more specific vulnerability like *Environment Configuration Issues*.

Regarding the structure itself, the taxonomy has a flat organization in which the categories do not really represent aspects at the same abstraction or conceptual level. For instance, *Denial of Service* is generally considered as a type of attack or the effect of an exploited vulnerability, or *Configuration Issues* is quite generic, and it does not characterize the vulnerability sufficiently. We can observe a similar issue between the names given to the categories and to the specific vulnerabilities, e.g., *Bad Coding Group* versus *Coding error Vulnerability*. Although the classification lists 54 vulnerabilities, it would benefit from evolving and including more recent ones (e.g., via an open repository).

Table 8 presents the classification proposed by Zhang et al. (2020a), in which faults are classified based on IEEE Standard Classification for Software Anomalies (I. Group et al 2010). The authors organized the faults into nine main groups: data, description, environment, interaction, interface, logic, performance, security, and standard. Thus, the paper does not focus on security vulnerabilities but on faults in general. In addition, despite the work listing 49 faults, some identified faults are more related to best practices and not to software defects. *Nonstandard naming*, *Wasteful contracts*, and *Specify Function Variable as Any Type* are examples of these types of faults presented in the paper. Also, some vulnerabilities could be placed in other groups. For instance, *Integer Truncation* is related to a cast operation and not to a calculation operation, and *Right-To-Left-Override Control Character* is related to a function Call with wrong arguments and not to "Description". Also, some vulnerabilities seem to be misplaced, such as *Results of Contract Execution Affected by Miners*, because it is a blockchain environment problem (i.e., a vulnerability in the miner process) and not related to programming. It is worthwhile mentioning that the paper could provide more details about each vulnerability/fault because some names (e.g., *Pre-sent Ether*) seem to be more related to normal operations than a vulnerability/fault. A code example for each classification entry would be helpful for this matter.

Chen et al. (2020a) present a taxonomy with 20 vulnerabilities (i.e., Table 9). The authors categorized the vulnerabilities according to potential security, availability, performance, maintainability, and reusability problems. Also, the paper proposes solutions (code examples) to avoid each vulnerability being activated. The taxonomy follows a flat structure that is somewhat confusing. Sometimes, an entry refers to a group of vulnerabilities (i.e., *Reentrancy*); sometimes, it refers to individual vulnerabilities (i.e., *Unspecified Compiler Version*). Some vulnerabilities such as *High Gas Consumption Function Type* and *High Gas Consumption Data Type* could be grouped inside a *Gas Consumption group*.

2.3 Community-Based Classification Schemes

This section discusses taxonomies or classification initiatives maintained by communities.

One of the most popular ones is Smart Contract Weakness Classification (SWC) (SmartContractSecurity 2020), a vulnerability classification scheme for smart contracts whose main goals are: i) Provide a straightforward way to classify 'weaknesses' of a smart contract; ii) Identify weaknesses that lead to vulnerabilities; iii) Define a common language to describe weaknesses in the architecture, design, and coding of smart contracts; and finally, iv) Being a way to improve the effectiveness of smart contract security analysis tools (Wagner 2018).

In SWC, each vulnerability has an external relationship with another taxonomy (i.e., CWE (CWE Community 2009)), and there are examples (i.e., faulty and non-faulty code) to illustrate the vulnerability and a correction. SWC has a flat list structure, where the distinction between vulnerabilities and other types of defects is often unclear. Also, it is worth mentioning that there are cases where it is difficult to distinguish whether the problem is related to the

Table 8 Vulnerability classification in Zhang et al. (2020a)

Group	Sub-Group	Vulnerability
Data	Calculation	Integer Division
		Integer Overflow and Underflow
		Integer Sign
		Integer Truncation
		Wrong Operator
	Hidden	Hidden Built-in Symbols
		Hidden State Variables
		Incorrect Inheritance Order
	Initialization	Unitialized Local/State Variables
		Unitialized Store Variables
Description	Output	Right-To-Left-Override Control Character
Environment	Supporting software	Delete Dynamic Array Elements
		Using continue statements in do-while statements
Interaction	Contract call	Re-entrancy Vulnerability
		Unhandled Exception
	Ether flow	Forced to receive ether
		Locked Ether
		Pre-sent Ether
Interface	Parameter	Call/delegatecall data/address is controlled externally
		Hash Collision with Multiple Variable Length Arguments
		Short Address Attack
		Signature with Wrong Parameter
	Token parameter	Nonstardard token interface
Logic	Assembly code	Returning results using assembly code in the constructor
		Specify Function Variable as Any Type
	Denial of service	DOS by Complexity Fallback Functions
		DOS by Gaslimit
		DOS by Non-existent address or malicious contract
	Fairness	Results of Contracts Execution Affected by Miners
		Transaction Order Dependence
	Storage	Storage Overlap Attack
Performance	Gas	Byte[]
		Invariants in Loop
		Invariants State Variables Are Not Declared Constant

Table 8 continued

Group	Sub-Group	Vulnerability
		Unused Public Functions Within The Contracts Should Be Declared External
Security	Authority control	Replay Attack
		Suicide Contracts
		Use tx.origin for authentication
		Wasteful contracts
		Wrong constructor name
	Privacy	Non-public variables are accessed by public/external functions
		Public Data
Standard	Maintainability	Implicity Visibility Level
		Nonstardard naming
		Too Many Digits
		Unlimited Compiler Versions
		Use Depracated Build-in Symbols
	Programming specification	View/constant function changes contracts states
		Improper use of require, assert and revert

Table 9 Classification proposed in Chen et al. (2020a)

Vulnerabilities
Unchecked external calls
Dos under external influence
Strict balance equality
Unmatched type assignment
Transaction state dependency
Reentrancy
Hard code address
Block info dependency
Nested call
Deprecated APIs
Unspecified compiler version
Misleading data location
Unused statement
Unmatched ERC-20 standard
Missing return statement
Missing interrupter
Missing reminder
Greedy contract
High gas consumption function type
High gas consumption data type

blockchain platform or to the smart contract itself (e.g., *Weak Sources of Randomness from Chain Attributes*, *Unencrypted Private Data On-Chain*). A positive aspect is that SWC is associated with an open repository, although, at the time of writing, the last update was made in 2018. Considering the changes and new knowledge about smart contract vulnerabilities, this means that practitioners' involvement is now impaired. For instance, the classification presented in Rameder et al. (2022) identifies several new vulnerabilities that are not present in SWC.

The NCC Group initiated the Decentralized Application Security Project (DASP) in 2018, which includes a vulnerability classification scheme for smart contracts. The main idea is to present the top 10 smart contract vulnerabilities, for which a single iteration was carried out precisely in 2018. Thus, it does not really reflect the whole landscape of vulnerabilities. DASP provides a short description for each class of vulnerabilities, which is accompanied by pseudo-code as a way of explaining the vulnerabilities in detail. The classification emphasizes the impact the vulnerability had in real-world scenarios (e.g., reentrancy loss estimated at 3.5M ETH 50M USD at the time). References to real-world attacks are provided (i.e., reports, magazines, etc.), which present a historical view of vulnerability exploitation. The nomenclature is clear, although some parts of the structure are questionable. For instance, the *Denial of Service* category in DASP refers to *gas limit reached*, *unexpected throw*, *unexpected kill*, and *access control breached*. The description is sometimes very short and may become ambiguous (e.g., *access control breached* may refer to a vulnerability that would simply fit in *Access Control*, which is another DASP category). In Durieux et al. (2020), the authors used DASP but concluded that the categories were not sufficient to cover the vulnerabilities found.

SIGP (Manning 2018) is a vulnerability classification scheme for smart contracts written in Solidity that forms the basis of of the work in Antonopoulos and Wood (2018). The classification considers three main elements: vulnerability, preventive technique, and a real-world example. The first element conceptually describes the reported vulnerability. It also presents the vulnerable code and explains how the attack is performed. The second element presents a solution for the problem, and the last element discusses a real-world attack in which the vulnerability was exploited. The clarity of the names used for the vulnerabilities could be improved (e.g., *entropy illusion* and *constructors with care* are ambiguous). There is an open repository associated, but not receiving any updates at the time of writing. As in previous cases, there are only 16 vulnerabilities listed, which is currently far from the state of the practice.

The SMARTDEC classification (SmartDec Corporation 2018) originated from the experience gathered from the creation of Smartcheck (Tikhomirov et al. 2018). The vulnerabilities are organized into three main categories: Blockchain (i.e., vulnerabilities from the blockchain system), Language (i.e., programming language defects), and Model (i.e., vulnerabilities caused by mistakes in the model). Each group has several entries (up to a total of 11), where each entry corresponds to a set of related vulnerabilities. The entry names are unique, although they are also quite generic and therefore less descriptive (e.g., *Trust*). The authors provide a mapping between their taxonomy and other classifications, namely DASP (NCCGroup 2021), SWC (SmartContractSecurity 2020), and SIGP (Manning 2018). As an example, the *Arithmetic* category is related to *Over/underflow* in SWC-101, DASP-3, and SP-2 and to *Precision issues* in SP-15. The repository is open to contributions, although, at the time of writing, there has been no update since 2018.

2.4 Classification Schemes in Vulnerability Detection Research

Research in smart contract vulnerability detection is generally accompanied by custom vulnerability classification schemes (Luu et al. 2016; Kalra et al. 2018; Wang et al. 2019; Ghaleb and Pattabiraman 2020; Choi et al. 2021; Bose et al. 2022). This is primarily due to lacking an appropriate and up-to-date classification standard or taxonomy. As a result, biased and limited classifications emerged, which are coupled to the context in which they were created. The next paragraphs describe the classification schemes of selected research, namely of three of the most cited vulnerability detection research works (at the time of writing and according to Google Scholar). In all of these cases, the heterogeneity is clear, as well as the divergence with other classification schemes, such as the ones previously presented in this section.

A symbolic execution tool named Oyente is proposed in Luu et al. (2016) with the goal of allowing practitioners to detect security vulnerabilities. In this work, the authors identify a small set of security vulnerabilities, as illustrated in Table 10.

Although the work in Oyente targets a specific set of vulnerabilities, the absence of a standard way for categorizing and naming the vulnerabilities impairs the assessment and comparison of results with other tools or approaches.

Securify (Tsankov et al. 2018) is a vulnerability detection tool based on symbolic execution methods, which, at the time of writing, is able to detect 37 security vulnerabilities (Tsankov 2018), grouped by severity, as we can see in Table 11.

Again, as with the previous tool, the groups and the names or vulnerability definition are non-standard, although there is an effort to classify most of them according to SWC (SmartContractSecurity 2020).

Zeus is a tool based on abstract interpretation and symbolic execution (Kalra et al. 2018). Table 12 shows the vulnerability classification performed by the authors and targeted by the tools.

As we can see in Table 12, the authors created several groups (e.g., incorrect contracts, unfair contracts), in which several vulnerabilities are placed. Although this is obviously a partial classification of known vulnerabilities, the heterogeneity of the naming and definitions and also general classification structures is relatively clear when compared to other works. It again emphasizes the need for a more standard way of categorizing vulnerabilities.

2.5 Limitations of Current Classification Schemes

In this section, we highlight the main gaps and limitations identified during the analysis of the different vulnerability classifications previously described, as follows:

– Classifications proposed in the literature tend to have simple structures, most of them simply grouping the vulnerabilities into related groups. Many times, no groups at all are used. This is not a problem by itself and may be useful in certain environments (e.g., a

Table 10 List of vulnerabilities in Luu et al. (2016)

Vulnerability name
Mishandled exceptions
Reentrancy
Timestamp dependence
Transaction-ordering dependence

Table 11 Vulnerability classification in Tsankov et al. (2018) and extended in Tsankov (2018)

Severity	Vulnerability
Critical	TODAmount
	TODReceiver
	TODTransfer
	UnrestrictedWrite
High	RightToLeftOverride
	ShadowedStateVariable
	UnrestrictedSelfdestruct
	UninitializedStateVariable
	UninitializedStorage
	UnrestrictedDelegateCall
	DAO
Medium	ERC20Interface
	ERC721Interface
	IncorrectEquality
	LockedEther
	ReentrancyNoETH
	TxOrigin
	UnhandledException
	UnrestrictedEtherFlow
	UninitializedLocal
	UnusedReturn
Low	ShadowedBuiltin
	ShadowedLocalVariable
	CallToDefaultConstructor?
	CallInLoop
	ReentrancyBenign
	Timestamp
Info	AssemblyUsage
	ERC20Indexed
	LowLevelCalls
	NamingConvention
	SolcVersion
	UnusedStateVariable
	TooManyDigits
	ConstableStates
	ExternalFunctions
	StateVariablesDefaultVisibility

Table 12 Vulnerability classification in Kalra et al. (2018)

Group	Vulnerability
Incorrect contracts	Reentrancy
	Unchecked send
	Failed send
	Integer overflow/underflow
	Transaction state dependence
Unfair contracts	Absense of logic
	Incorrect logic
	Logically correct but unfair
Miner's influence	Block state dependence
	Transaction order dependence

novice in the field of smart contracts trying to gain fast knowledge about security), however, such structures are often ad-hoc and, consequently, short-lived, resulting in limited adoption. The classifications that collect more vulnerabilities are found in Rameder et al. (2022); SmartContractSecurity (2020); Tsankov (2018), with Tsankov (2018) grouping vulnerabilities by criticality and with Rameder et al. (2022) using conceptual groups to fit related vulnerabilities.

- There is a large diversity of names being used in state of the art to refer to the same vulnerability (e.g., both *Integer bugs or arithmetic issues* and *Integer over or underflow* (Rameder et al. 2022) refer to the same vulnerability). There are also cases in which very similar names refer to different vulnerabilities (e.g., *unpredictable state* (Grishchenko et al. 2018) refers to wrong class inheritance order defect while *vulnerable state* (Krupp and Rossow 2018) refers to uninitialized storage variable defect). In some cases, the same name refers to different vulnerabilities, e.g., *Transaction Order Dependency (TOD)* is the name used in Liao et al. (2019) and in Bose et al. (2022), which refers respectively to "6.1.5 Transfer Amount Dependent on Transaction Order" and to "6.1.6 Transfer Recipient Dependent on Transaction Order" in OpenSCV.
- Current classifications include several generic names that do not assist in the classification of specific vulnerabilities (e.g., *call to the unknown* (Atzei et al. 2017) or *unexpected function invocation* (Chen et al. 2020b)). In several cases, unclear nomenclatures are used, such as *entropy illusion*, *constructors with care* (Manning 2018), or *improper Blockchain Magic Validation* (Amiet 2021), which do not specify what the vulnerability is. Another example is *Style guide violation* (Zhang et al. 2019), which is not even clear whether it is referring to bad practice or a vulnerability.
- Regarding vulnerability classification, current research appears to be falling far behind the state of the practice. Current vulnerability detection tools identify several vulnerabilities (e.g., Securify2 (Tsankov 2018)) that do not fit in relatively well-established classifications, such as DASP (NCC Group 2019), or SWC (SmartContractSecurity 2020).
- Current classifications do not involve active community participation, and we observed little to no participation at all in several classifications. Thus, it is fundamental that a classification can be easily maintained and evolve to integrate new vulnerabilities or even has the possibility of structurally changing (i.e., versioning is also required). This reduced community participation is the main reason why the most popular classification initiatives, like SWC (SmartContractSecurity 2020) or DASP (NCC Group 2019), are currently far behind the detection capabilities of vulnerability detection tools.

- Classifications originated from vulnerability detection tools sometimes use names that are biased towards the tool's capabilities, which is fully acceptable from a tool perspective, but for broader goals (e.g., tool benchmarking), a vulnerability classification must be independent of specific tools' capabilities. For instance, Osiris (Torres et al. 2018) is a tool for detecting vulnerabilities related to integer values and naturally focuses on a few types of issues affecting integer manipulation. Thus, the naming used is very specific to this context and also does not capture the larger picture (e.g., issues affecting other types of numbers may be related but are not represented).
- The high heterogeneity of names used across various tools, community efforts, and research initiatives creates a significant obstacle to understanding which tools perform better. Although initiatives exist to assess the effectiveness of vulnerability detection tools, they all face difficulties in adopting a uniform, fine-grained taxonomy for defects.
- Many times, taxonomies mix the characteristics of a certain vulnerability with the effect of exploiting it or with how it is exploited, or its impact, and use category names like *Denial of Service*, which is basically a consequence of the activation of a certain vulnerability. This is not necessarily wrong, but it may contribute to a non-uniform taxonomy and possibly error-prone from the point of view of the taxonomy's user. For instance, Ghaleb et al. (2023) classified the access control vulnerabilities into two vulnerabilities: *violated access control checks (VACC)* and *missing access control checks (MACC)*. The first one (i.e., VACC) refers to a vulnerability that results in improper access control. After analyzing the vulnerability presented in the paper, we realized that the attack occurred due to a wrong constructor name. The second one (i.e., MACC) refers to incorrect authentication or authorization. According to the paper (Ghaleb et al. 2023)), the vulnerability is activated when the developer makes a mistake in verifying the owner's identification (i.e., the Caller is not the owner). This clearly shows that the name selected for the vulnerability does not reflect the nature of the vulnerability but the impact of the vulnerability. OpenSCV covers both vulnerabilities within the entries *5.2.2 - Wrong Constructor Name* and *7.1.1 Wrong Caller Identification*, respectively.
- Classification structures are often constructed with different degrees of granularity. Some structures have general categories, while others have more specific categories. This inconsistent categorization poses difficulties and complexities for practitioners and tool developers, as they end up creating new classifications. Overall, a broader view of vulnerability detection is needed to foster the longevity of a particular taxonomy, accompanied by the possibility of evolving it.

3 OpenSCV Construction Process

This section describes the process followed to build the OpenSCV taxonomy. Overall, it was an iterative and incremental process during which we kept general taxonomy quality properties (e.g., the ones discussed in Section 2) in perspective while going through all the construction phases. As mentioned in Section 1, we use the general term *vulnerability* to refer to vulnerabilities and also to software defects considered in the literature to be associated with high-security risks. Figure 1 overviews the process, which consists of the following phases:

i) Vulnerability information collection;
ii) Vulnerability relationship with other classifications;
iii) Vulnerability characterization (defect type, qualifier, and code clip example);

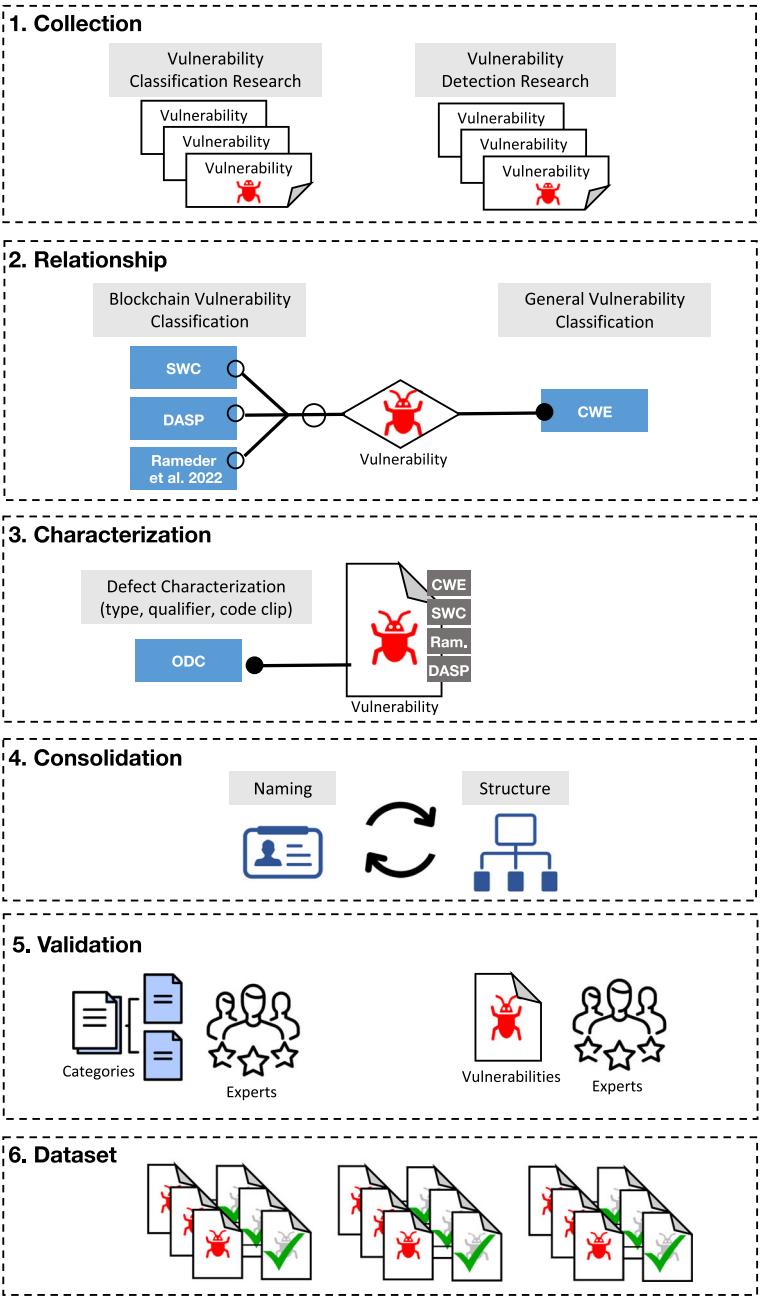

Fig. 1 Taxonomy construction process

iv) Structural and nomenclature consolidation;
v) Dataset construction.

Regarding the first phase (**vulnerability information collection**), visible on the top of Fig. 1), the main goal was to gather an up-to-date, heterogeneous, and non-curated list of vulnerabilities that affect smart contracts. This list allowed us to understand the naming and classification heterogeneity, which is essential to building an integrated vision and ultimately reaching a meaningful taxonomy.

To build our taxonomy, we mainly relied on two sets of works: i) the existing smart contract vulnerability classifications and ii) the smart contract vulnerability detection tools or works. We used Google Scholar to first identify research work on *smart contract vulnerability classification* (e.g., taxonomies and vulnerability classification schemes), which resulted in the identification of 11 research papers. We then proceeded to search for research targeting *smart contract vulnerability detection*, which resulted in the identification of 77 research papers, mostly materialized in tools and that we summarized in Table 13. The identified works refer to research carried out from October 2016 to April 2023 and resulted in the collection of 481 vulnerability definitions. To filter the identified studies, a set of inclusion and exclusion criteria was applied:

Inclusion Criteria:

a) The work must address smart contract vulnerability detection and, therefore, must specify types of vulnerabilities (e.g., *reentrancy* (Liu et al. 2018)).

b) The work must clearly characterize the technique used to perform vulnerability detection so that we can specify its type/subtype while allowing readers to understand its basic mechanics.

Exclusion Criteria:

a) Works that are published as short papers (e.g., less than 4 pages) are excluded from the analysis. The same happened with non-peer-reviewed research (i.e., pre-prints, despite being publicly available).

b) Due to the huge number of publications on this topic in recent years (i.e., detection verification tools) and also serving as a quality assessment measure, we exclude papers published in tier B conferences and lower (we use CORE 2021 (The Computing Research and Education Association of Australasia 2021), as reference), as well as papers published in JCR Q2 Journals and lower (Clarivate 2021). The exceptions to this are papers published in blockchain conferences or journals, which did not have enough time to enter the respective rankings and are considered due to this reason.

To go through the aforementioned criteria, we analyzed each paper's information, namely the title, abstract, and full text. The full-text analysis served initially to understand if vulnerability types are present and sufficiently described.

It is worth emphasizing that OpenSCV's primary focus is on the characterization and classification of security vulnerabilities within smart contracts, which may help developers of mitigation solutions for implementing more effective tools. Thus, the focus is not on threat mitigation, which can be found in other works, such as in Ivanov et al. (2023), where the authors classify threat mitigation solutions (e.g., vulnerability detection tools) for smart contracts across several dimensions. Also important to notice is the fact that the identified research led us to the identification of community-oriented initiatives, namely SWC (SmartContractSecurity 2020) and DASP (NCC Group 2019), SIGP (Manning 2018), and SMARTDEC (SmartDec Corporation 2018).

In the second phase, we analyzed the **vulnerability relationship with other classifications** by going through each of the identified vulnerabilities and mapping them to popular

Table 13 Vulnerability detection tools identified in the state-of-the-art

Categories	Tecnique	Tool name	Reference
Formal methods	Abstract interpretation	HFCCT	Li et al. (2022b)
		SoliDetector	Hu et al. (2023)
		Vulpedia	Ye et al. (2022)
		MadMax	Grech et al. (2020)
		Securify	Tsankov et al. (2018)
	Model checking	Crincoli et al	Crincoli et al. (2022)
		Helmholtz	Nishida et al. (2021)
		Zeus	Kalra et al. (2018)
		VeriSolid	Mavridou et al. (2019)
		SmartPulse	Stephens et al. (2021)
		VeriSmart	So et al. (2020)
	Symbolic execution	ConFuzzius	Torres et al. (2021)
		ExGen	Jin et al. (2023)
		GasChecker	Chen et al. (2021)
		HFCCT	Li et al. (2022b)
		MOPS	Fu et al. (2019)
		Pluto	Ma et al. (2022)
		SAILFISH	Bose et al. (2022)
		SmarTest	Sunbeom et al. (2021)
		sCompile	Chang et al. (2019)
		RA	Chinen et al. (2020)
		EthRacer	Kolluri et al. (2019)
		teEther	Krupp and Rossow (2018)
		Oyente	Luu et al. (2016)
		MAR	Wang et al. (2021)
		Zhang et al.	Zhang et al. (2022b)
		Osiris	Torres et al. (2018)
		DEPOSafe	Ji et al. (2020)
		Vultron	Wang et al. (2019)
	Theorem proving	Ayoade et al.	Ayoade et al. (2019)
Static code analysis	Pattern recognition	Vrust	Cui et al. (2022)
		SmartDagger	Liao et al. (2022)
		SmartCheck	Tikhomirov et al. (2018)
	Taint analysis	EOSIOAnalyzer	Li et al. (2022c)
		Sereum	Rodler et al. (2019)
		SmartDagger	Liao et al. (2022)
		SmartFast	Li et al. (2022d)
		Achecker	Ghaleb et al. (2023)
		Ethainter	Brent et al. (2020)
		Slither	Feist et al. (2019)
		EasyFlow	Gao et al. (2019)
		Osiris	Torres et al. (2018)

Table 13 continued

Categories	Tecnique	Tool name	Reference
		Clairyoyance	Xue et al. (2020)
		EthPloit	Zhang et al. (2020b)
Software testing	Concolic execution	EthRacer	Kolluri et al. (2019)
		ConFuzzius	Torres et al. (2021)
		Etherolic	Ashouri (2020)
	Fuzzing	SODA	Chen et al. (2020b)
		Gas Gauge	Nassirzadeh et al. (2023)
		Hfcontractfuzzer	Ding et al. (2021)
		DEPOSafe	Ji et al. (2020)
		xFuzz	Xue et al. (2022)
		Pied-Piper	Ma et al. (2023)
		ReDefender	Li et al. (2022a)
		Sereum	Rodler et al. (2019)
		SMARTIAN	Choi et al. (2021)
		SmartFuzzDriverGen	Pani et al. (2023)
		Solanalyser	Akca et al. (2019)
		GasFuzzer	Ashraf et al. (2020)
		ContractFuzzer	Jiang et al. (2018)
		sFuzz	Nguyen et al. (2020)
		Vultron	Wang et al. (2019)
		WASAI	Chen et al. (2022)
Machine learning	Classical learning	Eth2Vec	Ashizawa et al. (2021)
		xFuzz	Xue et al. (2022)
		MODNN	Zhang et al. (2022a)
		Peculiar	Wu et al. (2021)
		SmartDagger	Liao et al. (2022)
		SmartMix	Shakya et al. (2022)
		TMLVD	Zhou et al. (2022b)
		Slicing Matrix	Xing et al. (2020)
		Song et al.	Song et al. (2019)
		ContractWard	Wang et al. (2021)
	Deep learning	ASSBert	Sun et al. (2023)
		CodeNet	Hwang et al. (2022)
		DeeSCVHunter	Yu et al. (2021)
		EtherGis	Zeng et al. (2022)
		Gupta et al.	Gupta et al. (2022)
		ReVulDL	Zhang et al. (2022c)
		VSCL	Mi et al. (2021)
		Zhuang et al.	Liu et al. (2021)
		Zhuang et al.	Zhuang et al. (2020)
	Ensenble learning	Dynamit	Eshghie et al. (2021)

smart contract vulnerability classification schemes, namely SWC (SmartContractSecurity 2020) and DASP (NCCGroup 2021). We also selected, from the state of the art in vulnerability classification, what is, to the best of our knowledge, the currently largest and most recent vulnerability classification scheme proposed by Rameder et al. (2022). Then, we resorted to a broader security-related classification, namely the Common Weakness Enumeration (CWE) (CWE Community 2009), which provides us with a non-domain-specific view of each defect. Although the action consists of simply mapping vulnerabilities, it actually contributes to the characterization of each vulnerability. This may be useful for later taxonomy consolidation purposes (e.g., by merging defects that are the same but are represented with different names). Obviously, mapping the identified vulnerabilities to existing classifications also allows us to understand the exact coverage of existing classifications or disparities against the current state of the art or practice.

The third phase - **vulnerability characterization (defect type, qualifier, and code clip example)** - has the direct goal of detailing the vulnerability according to its nature and also by example, which allows for clarity of the explanation and may help in cases where the vulnerability description and remaining attributes are inadvertently left unclear. Regarding (*vulnerability nature*), we resort to the Orthogonal Defect Classification (ODC), namely to the 'defect type' attribute, which generally characterizes the type of software defect and can correspond to *Assignment/Initialization*, *Checking*, *Algorithm/Method*, *Function/Class/Object*, *Interface/O-O Messages*, *Timing/Serialization*, *Relationship* (IBM 2013b). We also make use of the 'defect qualifier' attribute, which characterizes the state of the implementation before a correction, namely if the defect refers to *missing*, *wrong*, or *extraneous* code. We also use the ODC extensions, as proposed in IBM (2013a), for vulnerabilities that relate to other aspects (e.g., defect types related to the process followed during compilation or management of libraries). For each vulnerability, we also extracted a code excerpt (when made available by the authors) that could represent the issue, as a way to reduce or eliminate any possible ambiguities that could still be present. For the cases where no code example was made available and the description allowed to build one, we created a Solidity code example as a way of further illustrating the defect. Thus, all of the identified vulnerabilities in OpenSCV are associated with a code example.

The fourth phase, **naming and structural consolidation**, consists of two steps: the attribution of names to the vulnerabilities, and ii) their organization in a tree structure. In the first step, we merged defects that referred to the same issue (despite being named differently by different authors. This required going through the names and descriptions of the different defects and, whenever provided by the authors, also analyzing the corresponding vulnerable code to understand if it referred to the same defect or not. The additional characterization (e.g., ODC) helped in such grouping. Obviously, during this step, several adjustments to the characterization of the defects were made, as well as corrections to the defects' relationships with other classifications. Figure 2 shows an example of the same vulnerability named differently by different authors. In Brent et al. (2018) named it *Unsecured Balance* (Fig. 2.a)) and it basically consists of a misnamed constructor while Zhang et al. (2019) named it *Missing constructor* (Fig. 2.b)), where we observe that it is actually a wrong name used during the definition of the constructor. So, besides the names, we actually see that the definitions provided may not be really accurate sometimes. In this particular case, and as an example, we named this defect as *Wrong Constructor Name*. Thus, during this step, we defined an initial name for each of the defects, based on the name given by the authors of the respective paper, on the names presented in the corresponding related classifications (i.e., DASP, SWC, CWE), and on the ODC classification.

a)

```
contract TaxMan {
    address private owner;
    ...
    function TaxOffice() {
        owner = msg.sender;
     }
     function collectTaxes() public {
         require(msg.sender == owner);
         owner.send(tax);}
}
```

b)

```
contract Foo
{
    address public owner;
    ...
    function foo() public
    {
        owner = msg.sender;
        ...
    }
}
```

Fig. 2 The same vulnerability named and also described differently in: a) (Brent et al. 2018); b) (Zhang et al. 2019)

Each vulnerability is analyzed and classified according to its most prevalent nature. For example, design flaws related to the lack of exception handling are linked to *Mishandled Events* category, or failures of functions that are known to be called externally but which need further checking are classified in *Unsafe External Calls* category in the OpenSCV.

In the second step of the fourth phase (i.e., structural consolidation), we defined a hierarchical structure for the taxonomy based on the merged vulnerabilities and preliminary naming. During this step, names were further adjusted for clarity and also to better fit in the categories being created. The final result is visible in Figs. 3 and 4. As we can see in both Figures, OpenSCV consists of three levels. The first one (at the left-hand side of both Figs. 3 and 4) contains the higher level categories, the intermediate nodes are hybrid and contains groups (i.e., subcategories) of vulnerabilities as well as a few isolated vulnerabilities. All items at the last level (leaves at the right-hand side of Figs. 3 and 4) represent vulnerabilities. Each vulnerability identified in the tree is labeled with several symbols that characterize it in terms of ODC defect type and ODC qualifier.

To build the taxonomy structure, we followed a bottom-up process and began by grouping the defects of similar nature, which allowed us to create a set of categories, such as *reentrancy*, *useless code*, or *improper type usage*, for example. Certain defects could not really be grouped, such as *Use of Malicious Libraries* or *Inadequate Data Representation*, although at the same time, many of them sounded like higher-level defects (i.e., siblings were expected). Thus, for the time being, we opted not to keep these vulnerabilities at the bottom layer (e.g., by creating a subcategory with a single vulnerability). After this, the same procedure was applied at this current intermediate level to reach the definition of the higher-level categories.

To build our taxonomy, we analyzed the nature of each vulnerability and the reason why each vulnerability may appear in the code. Although we provide information about the impact of each, the categorization and naming of the vulnerabilities is mainly based on the nature of each vulnerability (e.g., *Arithmetic Issues -> Division Bugs -> Divide by Zero*). The main reason for this is to help developers easily find the vulnerabilities and better understand how to avoid these vulnerabilities. Each leaf in our taxonomy belongs to a concrete vulnerability; thus, we provide concrete code examples for each.

The whole taxonomy construction process was iterative and required the involvement of 2 experienced researchers and 1 early stage researcher. During the process, several adjustments to names were performed for further clarity and consistency across all categories of the taxonomy. Obviously, this is a continuous effort, which is now open to community participation via our GitHub repository (Vidal et al. 2024b). The current shape of the taxonomy may evolve to incorporate further vulnerabilities. It is worth mentioning that, during this process, we

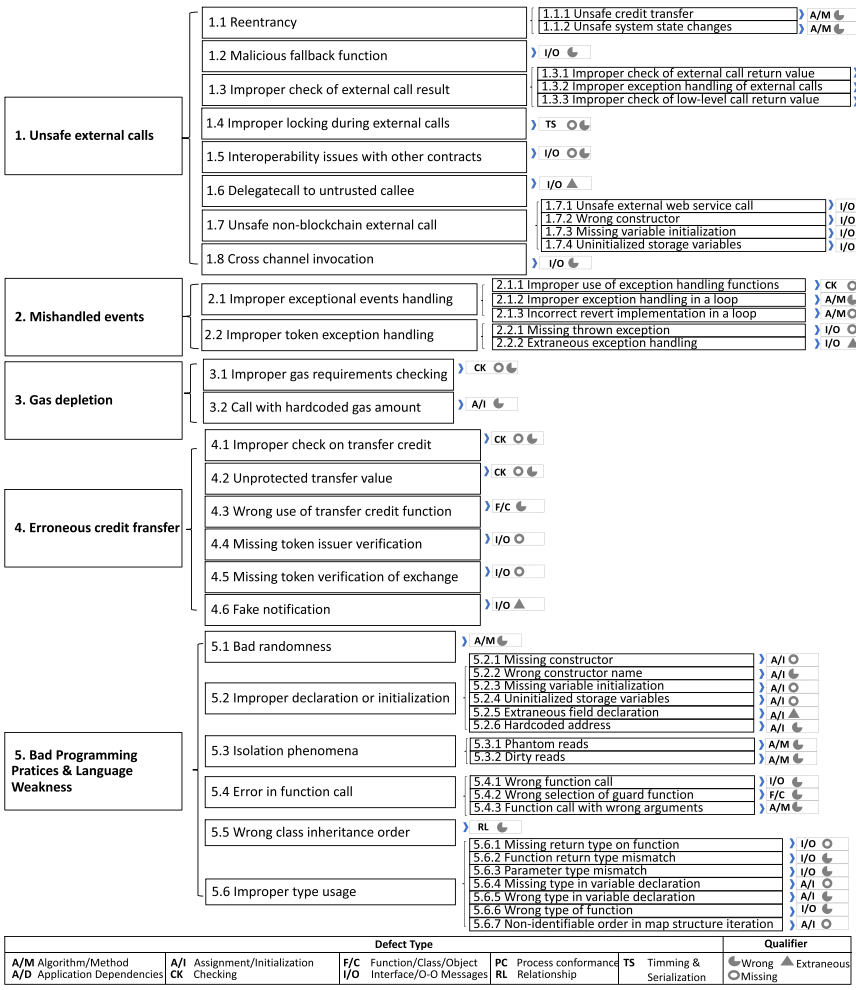

Fig. 3 Taxonomy of smart contract vulnerabilities (Part 1 of 2)

observed that the integration of new works on vulnerability detection was a major contributor to the definition of the taxonomy, and this is the reason why we intend to be continuously integrating new works on vulnerability detection and mapping their information into new versions of our taxonomy, possibly making naming and structural adjustments as a consequence of such integration. Currently, our taxonomy lists 94 vulnerabilities and is available at Vidal et al. (2024c), where all the mapped works are identified, as well as the vulnerability names used by those works. The taxonomy allows easy integration of new works. It is ready not only to support naming and structural changes but also to correct possible errors.

The fifth phase refers to the **validation** process in which we invited smart contract security experts to validate our taxonomy. They are mostly developers of vulnerability detection tools and are interested in this research area. Each participant was introduced to the vulnerabilities and their category individually and asked to evaluate the title and description of each vulnerability in terms of *Clarity* and *Comprehensibility* and also evaluate each category in terms of

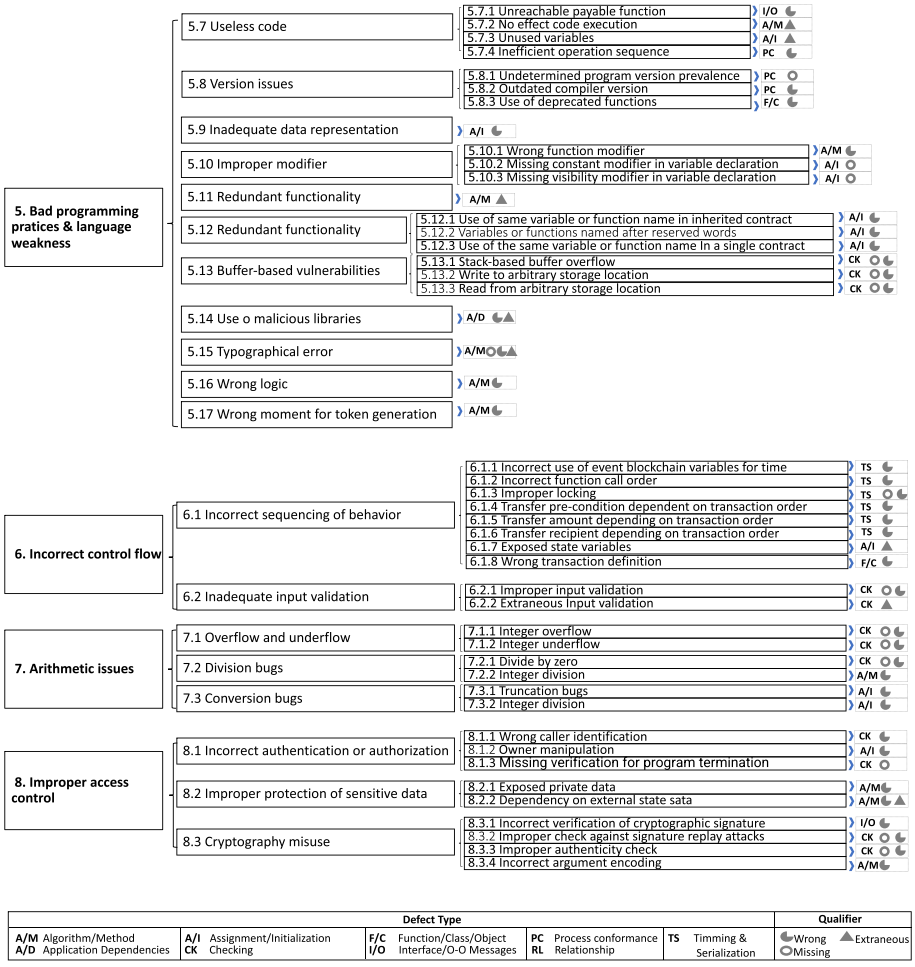

Fig. 4 Taxonomy of smart contracts' vulnerabilities (Part 2 of 2)

Representativeness, *Clarity*, *Comprehensiveness* and *Usefulness*. We also encourage them to provide additional feedback in order to improve the overall quality of the taxonomy.

The sixth phase refers to the **dataset constrution**, where we aimed at obtaining multiple real examples of smart contracts that match the defects present in our taxonomy. At the time of writing, a preliminary version of the dataset is created by gathering multiple real examples of contracts (i.e., a vulnerable contract and the corresponding correction) per each vulnerability presented in our taxonomy. Indeed, each vulnerability might appear in the code in different forms (i.e., different implementations), and vulnerability detection tools might be able to detect just some of the forms. We directly used examples from the collected papers whenever complete contracts were made available. In some cases, SWC also had good examples. All collected contracts present in our dataset passed through the compilation phase. Our intention is to provide an initial basis for researchers to use and, at the same time, provide the possibility of adding further examples (ideally, different forms of the same vulnerability being added to the dataset).

4 Taxonomy of Smart Contract Vulnerabilities

In this section, we traverse the taxonomy tree and briefly describe all categories and individual vulnerabilities. We use the original index numbers as presented in Figs. 3 and 4 (at the left-hand side of each taxonomy item) for consistency with OpenSCV's website (Vidal et al. 2024c), which provides a detailed description of the taxonomy. So, please note that the following item numbers (marked with the # symbol) refer to the index of a certain item in the taxonomy and not to subsections of the document.

To ensure the paper is kept within reasonable length, most of the descriptions consist of brief explanations. For further information, the reader may refer to Vidal et al. (2024c) and the provided examples. However, for illustrative purposes, we discuss the first vulnerability (i.e., *1.1.1 Unsafe credit transfer*)in detail.

1 Unsafe External Calls

This category represents a set of vulnerabilities in which there is an interaction between at least two contracts. It also refers to vulnerabilities related to contracts making non-blockchain external calls, such as calling external web services, calling external libraries, executing external commands, or accessing external files.

1.1 Reentrancy

The first subcategory is reentrancy, in which two or more contracts are involved: the vulnerable contract and the malicious contract. Overall, this type of vulnerability occurs when the malicious contract, after initiating a call, is allowed to make new calls to the vulnerable contract before the initial call has been completed. Thus, unexpected state changes may occur, such as depletion of credit. We identified two types of reentrancy vulnerabilities: one associated with loss of credit and the other associated with unexpected state changes. This is in line with several vulnerability detection tools, such as Securify (Tsankov 2018), Slither (Slither's Github 2019), or Momeni et al. (2019); Feist et al. (2019), which also distinguish these two cases (although using different names).

 # 1.1.1 Unsafe Credit Transfer

Known due to the DAO attack event (Siegel 2016), this vulnerability allows attackers to maliciously change balance via credit transfer calls that are allowed to take place before a previous call has been completed. Let us consider the case where a smart contract maintains the balance of several addresses, allowing the retrieval of funds. A malicious contract may initiate a withdrawal operation, which would lead the vulnerable contract to send funds to the malicious one before updating the balance of the malicious contract. Funds would be accepted on the malicious contract side, and a new withdrawal could be initiated (before the balance had been updated on the vulnerable contract side). As a consequence, the malicious contract could withdraw funds multiple times, with the total sum exceeding its own funds.

Using the Orthogonal Defect Classification (ODC) as a reference, this vulnerability can be classified as being of type *Algorithm* as the nature of the vulnerability sits in the logic created by the programmer. The ODC qualifier is defined as *wrong* as the vulnerability is related to incorrect logic (i.e., not *missing* or *extraneous* logic), related to the order of the instructions in the code. Figure 5 shows a concrete example available at OpenSCV's website (Vidal et al. 2024c) for this vulnerability.

Regarding the relationship to CWE, we classify this vulnerability (and actually the whole reentrancy group) as CWE-841, which describes a situation where "the software supports

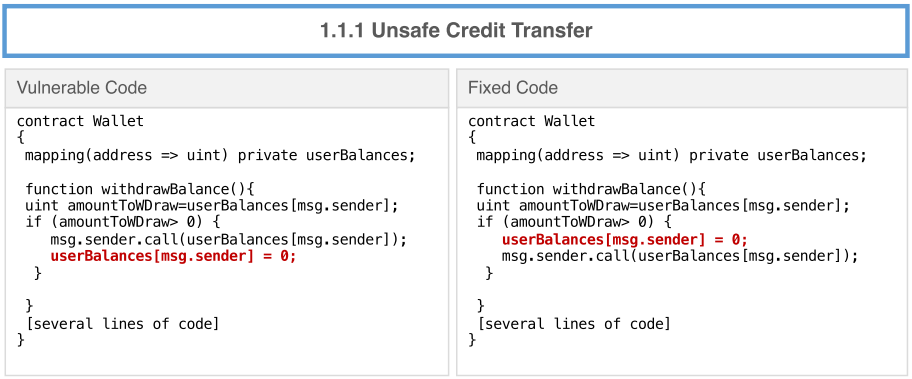

Fig. 5 Example for the *1.1.1 unsafe credit transfer* vulnerability at OpenSCV's website

a session in which more than one behavior must be performed by an actor, but it does not properly ensure that the actor performs the behaviors in the required sequence". This vulnerability is also known in the literature as *simple reentrancy* (Li et al. 2022a), *modified reentrancy* (Li et al. 2022a), *reentrancy-eth* (Li et al. 2022d), *cross-function re-entrancy* (Mavridou et al. 2019), *DAO* (Tsankov et al. 2018), *buggy-locked reentrancy* (Li et al. 2022a), *reentrancy vulnerabilities* (Chinen et al. 2020), *reentrancy* (Gupta et al. 2022; Sun et al. 2023; Chen et al. 2020b; Hwang et al. 2022; Yu et al. 2021; Eshghie et al. 2021; Ashizawa et al. 2021; Zeng et al. 2022; Hu et al. 2023; Xue et al. 2022; Zhang et al. 2022a; Fu et al. 2019; Ma et al. 2022; Wu et al. 2021; Rodler et al. 2019; Bose et al. 2022; Choi et al. 2021; Liao et al. 2022; Zhou et al. 2022b; Mi et al. 2021; Ye et al. 2022; Liu et al. 2021; Zhuang et al. 2020; Ashouri 2020; Ashraf et al. 2020; Feist et al. 2019; Jiang et al. 2018; Kalra et al. 2018; Luu et al. 2016; Wang et al. 2021; Mavridou et al. 2019; Nguyen et al. 2020; Stephens et al. 2021; Song et al. 2019; Tikhomirov et al. 2018; Wang et al. 2021, 2019; Xue et al. 2020; Akca et al. 2019; Torres et al. 2021; Zhang et al. 2022c), or *SWC-107: reentrancy* (SmartContractSecurity 2020).

 # 1.1.2 Unsafe System State Changes

This vulnerability is similar in nature to *v1.1.1*, with the main difference being the fact that there is no credit involved and, thus, no impact on users' funds. Due to the way the contract is coded, a call that reaches the vulnerable contract before a previous one has ended may allow an attacker to place the program in an unexpected state, leading to various effects, depending on the type of contract involved, including performance or availability issues. This vulnerability is also known in the literature as *reentrancy Benign* (Tsankov et al. 2018), *reentrancy No ETH* (Tsankov et al. 2018), or *SWC-107: reentrancy* (SmartContractSecurity 2020).

1.2 Malicious Fallback Function

Fallback functions are functions that are executed when a program receives a call to a function whose signature does not exist, i.e., either the name does not exist, or the parameters do not match the parameters of any of the existing functions. An attacker could exploit a smart contract through a vulnerability fallback function externally, which could activate other vulnerabilities (i.e., *Reentrancy* or *Access Control*). For instance, the attacker could invoke it

and reach a state that was not allowed to reach (Chen et al. 2020b). This vulnerability is also known in the literature as *call to the unknown* (Atzei et al. 2017) or *unexpected function invocation* (Chen et al. 2020b)

1.3 Improper Check of External Call Result

This category groups vulnerabilities that verify the execution of external contracts in an improper manner (i.e., verification is wrong or even missing), which affects the subsequent logic of the calling contract. The result of invoking a certain external operation should be verified, first of all, because it may simply fail, but especially because the called operation may be malicious (or may just have been poorly coded, resulting in an unexpected result); thus, the direct use of the result may lead to unexpected behavior.

 # 1.3.1 Improper Check of External Call Return Value

This vulnerability consists of an incorrect (or missing) verification of the returned value from the external execution of a contract. When a smart contract invokes another one, the returned value should be verified because the called operation may return an unexpected value (i.e., either because the callee is malicious or may just have been poorly coded, resulting in an unexpected result) (Chen et al. 2020b). This vulnerability is also known in the literature as *call-stack depth attack* (Song et al. 2019; Wang et al. 2021), *call depth* (Sun et al. 2023; Zhang et al. 2022a), *no check after contract invocation* (Chen et al. 2020b), *unchecked call return value* (Zheng et al. 2021), *unchecked external call* (Tikhomirov et al. 2018), *unused Return* (Tsankov et al. 2018), *unchecked return values* (Fu et al. 2019), or *SWC-104: Unchecked Call Return Value* (SmartContractSecurity 2020).

 # 1.3.2 Improper Exception Handling of External Calls

In the case of this vulnerability, the problem resides in the incorrect (or missing) handling of exceptional behavior thrown by a call (i.e., instead of residing in the handling of values, as in the case of vulnerability *v1.3.1*). The improper verification of exceptions thrown by the callee may lead to unexpected behavior in the caller contract. There are various reasons why the callee may exhibit exceptional behavior. For instance, the callee could be under malicious control, the execution of the transaction could activate a fault in the callee contract, the transaction could be terminated due to reaching the gas limit, or the callee contract may have been terminated (e.g., after a software fault has been detected in the contract). This vulnerability is also known in the literature as *denial of service* (Ashouri 2020), *doS by external contract* (Tikhomirov et al. 2018), *dos attack* (Liao et al. 2022), *external contract referencing* (Mavridou et al. 2019) or *SWC-113: DoS with Failed Call* (SmartContractSecurity 2020).

 # 1.3.3 Improper Check of Low-Level Call Return Value

Languages like Solidity provide the possibility of using low-level calls that operate over raw addresses. Such calls do not verify that the code exists or the success of the calls. Thus, its use may lead to unexpected behavior (Xi and Pattabiraman 2023). As a result, using such calls can be risky and should be avoided in most cases. This vulnerability is also known in the literature as *check effects* (Zhang et al. 2022a), *inline assembly* (Zhang et al. 2022a), *low level calls* (Tsankov et al. 2018; Zhang et al. 2022a), *unchecked call* (Hu et al. 2023), *unchecked low-level calls* (Hwang et al. 2022), or *low-level-calls* (Li et al. 2022d).

This issue has been addressed in the latest Solidity compiler, version 0.8.20, at the time of writing. If encountered, the compiler provides the following informative warning message: "Warning: Return value of low-level calls not used".

1.4 Improper Locking During External Calls

A vulnerable contract uses a lock mechanism in an erroneous manner, which may cause deadlocks. For instance, when a developer creates a condition with strict equality checks for the Ether balance, this condition can never be satisfied, which could allow an attacker to explore a DOS attack. This may result, for instance, in the impossibility of executing transfers and eventually in Denial of Service (Mavridou et al. 2019). This vulnerability is also known in the literature as *SWC-132: Unexpected Ether balance* (SmartContractSecurity 2020).

1.5 Interoperability Issues with Other Contracts

This issue relates to interoperability issues between contracts built in different language versions. Newer contracts may execute or inherit discontinued functionality present in older contracts (Khan et al. 2021). For instance, Solidity has introduced the operation code `STATICCALL` to allow a contract to call another contract (or itself) without modifying the state. Starting from V0.5.0, *pure* and *view* functions must now be called using the code `STATICCALL` instead of the usual `CALL` code. Consequently, when defining an interface for older contracts, the programmer should only use *view* instead of constant in the case s/he is absolutely sure that the function will work with `STATICCALL` (Solidity 2023). This vulnerability is also known in the literature as *assembly Usage* (Tsankov et al. 2018).

1.6 Delegatecall to Untrusted Callee

Calling untrusted contracts using the delegate feature is generally highly problematic because it opens the possibility for the called contract to change sensitive variables (e.g., `msg.data` or `sender`) of the source contract (Jiang et al. 2018). This type of issue has been most notably known as the Parity hack, which allowed attackers to reset the ownership and usage arguments of existing user wallets (Krupp and Rossow 2018). This vulnerability is also known as *code injection* (Krupp and Rossow 2018), *control-flow hijack* (Choi et al. 2021), *cross program invocation* (Cui et al. 2022), *dangerous delegate call* (Ashraf et al. 2020; Jiang et al. 2018), *delegate call* (Hu et al. 2023; Zeng et al. 2022; Xue et al. 2022), *delegate call abuse* (Fu et al. 2019), *tainted delegate call* (Brent et al. 2020), *unchecked delegate call function* (Nguyen et al. 2020), *unrestricted delegate call* (Tsankov et al. 2018), *unsafe delegate call* (Torres et al. 2021), or *SWC-112: Delegate call to Untrusted Callee* (SmartContractSecurity 2020).

1.7 Unsafe Non-Blockchain External Call

This category of vulnerabilities is related to contracts i) making non-blockchain external calls to untrusted third-party resources like web services and libraries, ii) executing external commands, or iii) accessing external files. These calls or accesses to non-blockchain external resources will cause security issues if different results are returned to each peer node, which consequently will result in inconsistent endorsements.

 # 1.7.1 Unsafe External Web Service Call

Sometimes, contracts may opt to streamline the development process and reduce effort by reusing external web services through API calls. Nevertheless, this approach can potentially lead to security issues if the returned values vary across different peer nodes. This vulnerability is also known in the literature as *Web service* (Li et al. 2022b).

 # 1.7.2 Unsafe External Library Call

Similar to the previously mentioned vulnerability, contracts might incorporate external libraries without a comprehensive understanding of their internal workings. This vulnerability is also known in the literature as *External Library Calling* (Li et al. 2022b).

 # 1.7.3 Unsafe External Command Execution

External command execution is another possibility within smart contracts. Nevertheless, this action cannot guarantee consistent results across all peer nodes. This vulnerability is also known in the literature as *System Command Execution* (Li et al. 2022b).

 # 1.7.4 Unsafe External File Access

Like external command execution, smart contracts also enable access to external files, but it cannot be assured that the nodes will receive identical results. This vulnerability is also known in the literature as *External File Accessing* (Li et al. 2022b).

 # 1.8 Cross Channel Invocation

Certain blockchain platforms like Hyperledger Fabric permit contracts to call each other. However, when two contracts interact through different channels, inconsistencies can arise in message reception. This vulnerability is known in the literature with the same name (Li et al. 2022b).

 # 2 Mishandled Events

This category includes a set of vulnerabilities in which exceptional events are mishandled. In Solidity, specific functions can be used to verify if certain conditions exist and throw exceptions in case the conditions are not met, namely `require` and `assert`. There are, however, fundamental differences. When the `require` function returns false, all executed changes are reverted, and all remaining gas fees are refunded. When the `assert` function returns false, it reverts all changes but consumes all remaining gas. However, such differences have become a frequent source of problems (Hajdu and Jovanović 2020).

 # 2.1 Improper Exceptional Events Handling

This first group of vulnerabilities is directly related to exceptional events, which, when mishandled, are often linked to the loss of atomicity in operations and other effects, such as excessive gas consumption or unauthorized access.

 # 2.1.1 Improper Use of Exception Handling Functions

Diverse run-time errors (e.g., out-of-gas error, data type overflow error, division by zero error, array-out-of-index error, etc.) may happen after a compiled smart contract is deployed. However, Solidity has many functions for error handling (e.g., `throw`, `assert`, `require`, `revert`), but their correct use relies on the experience and expertise of the developer. This vulnerability occurs when the developer misuses the handling exception functions, which can lead the program to unexpected behavior. This vulnerability is also known in the literature as *exception disorder* (Jiang et al. 2018), *exception state* (Zhou et al. 2022b), *mishandled exceptions* (Choi et al. 2021; Fu et al. 2019; Luu et al. 2016; Mavridou et al. 2019; Nguyen et al. 2020), *unexpected revert* (Ye et al. 2022), *unhandled errors* (Li et al. 2022b), *unhandled exception* (Ashouri 2020; Torres et al. 2021), or *unhandled exception* (Tsankov et al. 2018).

2.1.2 Improper Exception Handling in a Loop

This vulnerability occurs when a transaction is excessively large (i.e., it executes too many statements) and may lead to excessive costs. For instance, when one of the statements in a transaction fails (e.g., due to a software bug), the transaction will not be packaged into a block, and the consumed gas will not be returned to the user (and actually the concluded operations are reverted and must be executed again). Thus, such kinds of transactions should be decomposed into smaller parts so that the likelihood of success increases and the negative effects associated with the failure cases diminish. This vulnerability is also known in the literature as *call in loop* (Tsankov et al. 2018), *costly loop* (Shakya et al. 2022; Tikhomirov et al. 2018), *expensive operations in a loop* (Chen et al. 2021), *fusible loops* (Chen et al. 2021), *repeated computation in a loop* (Chen et al. 2021), *revert DOS* (Stephens et al. 2021), *unilateral comparison in a loop* (Chen et al. 2021), *unbounded mass operation* (Grech et al. 2020), *costly-operations-loop* (Li et al. 2022d), *gas limit DoS on a contract via unbounded operations* (Nassirzadeh et al. 2023), or *SWC-128: DoS With Block Gas Limit* (SmartContractSecurity 2020).

2.1.3 Incorrect Revert Implementation in a Loop

In the case of this vulnerability, the developer incorrectly specifies how the revert operation should be handled (in the context of a loop or a transaction composed of multiple operations), which ends up in a partial revert of the whole set of operations that should be reverted. This vulnerability is also known in the literature as *non-isolated calls (wallet griefing)* (Grech et al. 2020), *push DOS* (Stephens et al. 2021), or *SWC-126: Insufficient Gas Griefing* (SmartContractSecurity 2020).

2.2 Improper Token Exception Handling

The ERC-20 standard (Vogelsteller and Buterin 2015) provides functionalities to exchange tokens. Besides describing the functionalities, the standard specifies good practices for developers to implement its features. Regarding the `transfer` function, exceptional events can become problematic if handled improperly.

2.2.1 Missing Thrown Exception

Regarding the `transfer` function (i.e., functionality to transfer tokens from one account to another), the ERC-20 standard recommends that the developer throw an exception when a condition of the caller's account balance does not have enough tokens to spend. This allows the caller to understand the reason for which the transfer is not completed and take appropriate action. This vulnerability is also known in the literature as *ERC-20 transfer* (Ashizawa et al. 2021), *missing the transfer event* (Chen et al. 2020b), or *non-standard implementation of tokens* (Ji et al. 2020).

2.2.2 Extraneous Exception Handling

This type of vulnerability refers to the implementation of extra actions compared to what is recommended in a certain specification. The specification does not recommend actions like using guard functions (e.g., require or assert) in addition to throwing an exception when there is no balance in the caller. The extra actions might be arbitrary and incompatible with the purpose of a transfer functionality (e.g., returning true or false to report the success of the execution). This vulnerability is also known in the literature as *flawed back-end Verification of CEXes* (Ji et al. 2020), *infinite loop* (Liu et al. 2021; Zhuang et al. 2020), or *token API violation* (Tikhomirov et al. 2018).

 # 3 Gas Depletion

This category groups vulnerabilities that, in different ways, lead to gas depletion of the account used for the smart contract execution.

 # 3.1 Improper Gas Requirements Checking

This vulnerability represents missing or wrong checking of the prerequisites (i.e., in terms of gas) for executing a certain operation, causing unnecessary processing and use of memory resources. For cost management reasons, languages offer programmers several ways to deal with the cost of executing a certain operation in a contract. For instance, for transferring credits, Solidity provides the functions `transfer()` and `send()`, which have a limit of 2300 gas units for each execution. An alternative is to build a custom transfer function, where the gas limit is defined by a variable (e.g., `address.call.value(ethAmount).gas(gasAmount)()`). Despite having several ways of managing the program costs, it is challenging for programmers to predict which part of the code may fail. If an out-of-gas exception is triggered, the result may be unexpected behavior. This vulnerability is also known in the literature as *extra gas consumption* (Shakya et al. 2022), *gas consumption* (Ashizawa et al. 2021), *gas Dos* (Stephens et al. 2021), *gas less send* (Ashraf et al. 2020; Nguyen et al. 2020; Jiang et al. 2018; Wang et al. 2019), *opaque predicate* (Chen et al. 2021), *out of gas* (Akca et al. 2019), or *SWC-126: Insufficient Gas Griefing* (SmartContractSecurity 2020).

 # 3.2 Call with Hardcoded Gas Amount

This vulnerability refers to the impossibility of adjusting the amount of gas a certain program uses after being deployed. This issue is related to the observation that certain credit transfers in real contracts were being deployed using a fixed amount of gas (i.e., 2300 gas). If the gas cost of EVM instructions changes during, for instance, a hard fork, previously deployed smart contracts will easily break. This vulnerability is also known in the literature as *SWC-134: Message call with hardcoded gas amount* (SmartContractSecurity 2020).

 # 4 Erroneous Credit Transfer

This category groups vulnerabilities that are generally related to improper credit transfer operations.

 # 4.1 Improper Check on Transfer Credit

This vulnerability refers to the absence of verification (or wrong verification) after a transfer event, which can lead to an erroneous vision of the correct balance of the account. Indeed, the balance of the account may not reflect the currency transferred in an exact manner, leading to potential errors and opening the door to security issues. This vulnerability is also known in the literature as *forged transfer notification* (Li et al. 2022c), *unchecked send* (Kalra et al. 2018; Akca et al. 2019; Stephens et al. 2021), or *including Fake EOS transfer* (Li et al. 2022c).

This issue was addressed in the latest Solidity compiler, version 0.8.20, at the time of writing. If encountered, the compiler provides the following informative warning message: "Warning: Failure condition of send ignored. Consider using transfer instead".

4.2 Unprotected Transfer Value

The `transfer` function uses a numeric variable for transfers and may be vulnerable if it does not protect or specify limits for the values. When attribute `address.balance` is used for identifying the amount to be transferred, it will result in transferring the total balance at once, which is a high-risk operation for the cases where the amount is high (Zhang et al. 2020b). This vulnerability is also known in the literature as *arbitrarily transfer* (Ma et al. 2023), *ETH transfer inside the loop* (Shakya et al. 2022), *ether leak* (Choi et al. 2021), *manipulated balance* (Hu et al. 2023), *multiple send* (Choi et al. 2021), *transfer forwards all gas* (Tikhomirov et al. 2018), *unchecked transfer value* (Zhang et al. 2020b), *unrestricted ether flow* (Tsankov et al. 2018), or *SWC-105: Unprotected Ether Withdrawal* (SmartContractSecurity 2020).

4.3 Wrong use of Transfer Credit Function

Depending on the programming language, there are different ways to carry out credit transfer operations. In Solidity, `transfer` and `send` will both allow executing a credit transfer. However, in the case of a problem, `transfer` will abort the process with an exception, whereas `send` function will return `false`, and transaction execution is continued. An attacker may manipulate the `send` function and be able to continue executing a credit transfer operation without proper authorization. This vulnerability is also known in the literature as *failed send* (Kalra et al. 2018, *send instead of transfer* (Shakya et al. 2022; Tikhomirov et al. 2018)

4.4 Missing Token Issuer Verification

This vulnerability is related to EOSIO blockchain, in which the ‘transfer‘ function allows attackers to win the cryptocurrency without paying a ticket fee. This vulnerability is also known in the literature as *fake EOS* (Chen et al. 2022).

4.5 Missing Token Verification of Exchange

This vulnerability arises when an attacker can perform a fake deposit due to inadequate verification in the exchange implementation, specifically when unsafe usage of `transfer` or `transferFrom` functions is present. A potential solution for this issue involves adopting the `safeTransferFrom` function, which incorporates security checks before invoking the `transferFrom`, thereby mitigating the risk. This vulnerability is also known in the literature as *flawed token Verification of DEXes* (Ji et al. 2020).

4.6 Fake Notification

This vulnerability is related to the EOSIO blockchain, specifically in EOS notifications. The problem occurs when the attackers forward the notification from eosio.token to the victim and forge an EOS notification. This vulnerability is also known in the literature as *fake notification* (Chen et al. 2022) or *fake receipt* (He et al. 2021).

5 Bad Programming Practices and Language Weaknesses

This category represents issues that are mostly related to bad programming practices (i.e., error-prone or insecure coding practices) and language weaknesses, which are mostly related to insufficient protection mechanisms offered by the language, allowing the developers to make mistakes that could be avoided, e.g., by language constructs.

5.1 Bad Randomness

This vulnerability is related to using the variables that control the blocks in a blockchain to generate randomness, which is not secure. Such variables may be manipulated by miners so that the randomness is subverted, compromising the security of the blockchain, with its information becoming vulnerable to attacks. In fact, generating a strong enough source of randomness can be very challenging. The use of variables like `block.timestamp`, `blockhash`, `block.difficulty`, and other fields is problematic as these can be manipulated by miners. For example, a miner could select a specific timestamp within a delimited range or use powerful hardware to mine several blocks quickly, choose the block that would provide an interesting hash, and drop the remaining. This vulnerability is also known in the literature as *bad randomness* (Hu et al. 2023; Crincoli et al. 2022; Ashouri 2020), *bump Seeds* (Cui et al. 2022), *generating randomness* (Gupta et al. 2022), *random number generation* (Li et al. 2022b), *use predictable variable* (Zhou et al. 2022b), *predicable variable dependency* (Fu et al. 2019), or *SWC-120: Weak Sources of Randomness from Chain Attributes* (SmartContractSecurity 2020).

5.2 Improper Declaration or Initialization

The smart contract has resources that are either not initialized or initialized incorrectly, leading to unexpected behavior.

 # 5.2.1 Missing Constructor

A smart contract constructor is a function that is executed exactly once during the lifetime of a contract. It executes at deployment time, initializes state variables, performs a few necessary tasks that the specific contract requires, and sets the contract owner. If there is no constructor, the developer will have to implement such tasks manually, which is prone to security issues (e.g., variables may be set with incorrect values or forgotten, which may result in security problems). This vulnerability is also known in the literature as *SWC-118: Incorrect Constructor Name* (SmartContractSecurity 2020).

 # 5.2.2 Wrong Constructor Name

This vulnerability is related to the contracts that were published without a constructor because the programmer created a function, imagining it would behave like a constructor. Usually, the construction function has sensitive code (e.g., assignment of the owner of the contract), and by declaring a wrong function name, any user can call the function, thus, causing serious security risks. This vulnerability is also known in the literature as *erroneous constructor name* (Hu et al. 2023), *violated access control checks (VACC)* (Ghaleb et al. 2023), or *SWC-118: Incorrect Constructor Name* (SmartContractSecurity 2020).

This issue has been addressed in the latest Solidity compiler, version 0.8.20, at the time of writing. If encountered, the compiler provides the following informative error message: "Error: Functions are not allowed to have the same name as the contract. If you intend this to be a constructor, use constructor(...) ... to define it. Error: Expected identifier but got ()".

 # 5.2.3 Missing Variable Initialization

This vulnerability refers to the lack of initialization of variables that are used throughout the contract. Obviously, the effects can largely vary, depending on the variable itself and on the context in which it is being used. This vulnerability is also known in the literature as *golang grammar error* (Li et al. 2022b), *uninitialized variables* (Feist et al. 2019), *uninitialized-local* (Tsankov et al. 2018), or *uninitialized state variable* (Tsankov et al. 2018).

 # 5.2.4 Uninitialized Storage Variables

In Solidity, state variables are assigned to memory or storage. When a state variable is declared, it is assigned to a certain storage slot. If that variable is not initialized, it will be stored in slot 0 (the first one) of the contract's storage. Thus, it may conflict with the next variable that is declared in the same slot, causing an address conflict. This latter variable will overwrite the first, leading to unexpected behavior. This is the reason why it is important to initialize all state variables in a smart contract so that they are set into the correct storage slots (and possible conflicts are avoided) (Antonopoulos and Wood 2018). This vulnerability is also known in the literature as *uninitialized storage pointers* (Antonopoulos and Wood 2018), *uninitialized Storage* (Tsankov et al. 2018; Hu et al. 2023), or *SWC-109: Uninitialized Storage Pointer* (SmartContractSecurity 2020).

This issue has been addressed in the latest Solidity compiler, version 0.8.20, at the time of writing. If encountered, the compiler provides the following informative error message: "Error: This variable is of storage pointer type and can be accessed without prior assignment, which would lead to undefined behavior.".

 # 5.2.5 Extraneous Field Declaration

This vulnerability occurs when the programmer leaves field declarations in the contract structure, thereby enabling direct access to these fields (i.e., as they are defined in the structure). Given the possibility of the node environment falling out of sync, the contract's field values may diverge and become inconsistent among peer nodes. This vulnerability is also known in the literature as *field declarations* (Li et al. 2022b).

 # 5.2.6 Hardcoded Address

This vulnerability occurs when the programmer writes the code with a static address that should be an input variable. This vulnerability is also known in the literature as *hardcoded address* (Shakya et al. 2022)

5.3 Isolation Phenomena

This category gathers vulnerabilities that occur due to blockchain synchronization systems (i.e., enforced by consensus mechanisms), which can lead a program to produce different results at different times for the same query.

 # 5.3.1 Phantom Reads

The Hyperledger Fabric provides mechanisms for reading the ledger (i.e., `getPrivateDataQueryResult`), similar to querying conventional databases, but with the difference that the programmer cannot decide about the isolation level of the ledger. Thus, a contract can read out-of-sync node information from the ledger (i.e., during the validation phase), computing and/or processing based on outdated information. This vulnerability is also known in the literature as *range query risks* (Li et al. 2022b).

 # 5.3.2 Dirty Reads

In the context of HF, a query may return a key value before its update within the same transaction. As a consequence, this behavior can lead to unexpected results, as the returned value might not reflect the most recent update. This vulnerability is also known in the literature as *read-write conflict* (Li et al. 2022b).

5.4 Error in Function Call

In a blockchain context, each function in a smart contract is identified by its name, input parameters, and output parameters. Thus, these items compose the function *signature*, which is used by the contracts to verify that the right function is being called. This category groups vulnerabilities in which a developer uses a function in the wrong manner: either a wrong signature is used, wrong arguments are used, or a wrong function is called.

 # 5.4.1 Wrong Function Call

The issue occurs when a contract executes a certain function at a wrong address, i.e., at the address used by another function, which, however, has the same signature as the intended function. This vulnerability is also known in the literature as *type casts* (Atzei et al. 2017).

 # 5.4.2 Wrong Selection of Guard Function

`Assert` is a Solidity function, which is recommended to be used only in the development phase. Intentionally, the programmer inserts the function at a specific point in the program where a bug is suspected. If running the program results in gas depletion, the suspicion is confirmed.

Thus, this vulnerability refers to cases where the `assert` function is implemented with the wrong purpose, not having the expected effect. In more severe cases, in which the programmer forgets to remove it from the code or does not replace it with `require`, the impact of this vulnerability can be serious. This vulnerability is also known in the literature as *assert fail* (Zhang et al. 2022a), *assertion failure* (Choi et al. 2021; Torres et al. 2021), *assertion violation* (Sunbeom et al. 2021), or *SWC-110 Assert Violation* (SmartContractSecurity 2020).

 # 5.4.3 Function Call with Wrong Arguments

This vulnerability refers to the presence of certain control characters within the arguments of a function call, namely the *right-to-left override control character*, which can cause the function to execute with arguments in reverse order. This is a known issue also in other computing areas (Yosifova and Bontchev 2021). This vulnerability is also known in the literature as *right to left override* (Tsankov et al. 2018), *rtlo* (Li et al. 2022d), or *SWC-130: Right-To-Left-Override control character (U+202E)*.

This issue has been addressed in the latest Solidity compiler, version 0.8.20, at the time of writing. If encountered, the compiler provides the following informative error message: "Error: Mismatching directional override markers in a comment or string literal".

5.5 Wrong Class Inheritance Order

Contracts may have inheritance relationships with other contracts. In the case of solidity, the code of the inherited contract is always executed first, e.g., so that state variables are initialized properly. Solidity uses an algorithm named C3 linearization to determine the order in which the contracts are to be executed. Developers specify the inheritance relationships in a `inherit` statement and may believe that the order in which the inherited contracts are specified in that statement reflects the order in which the linearization algorithm should work. This opens space for security issues due to the wrong order of the contract in the `inherit` statement. This vulnerability is also known in the literature as *SWC-125: Incorrect Inheritance Order* (SmartContractSecurity 2020).

This issue has been addressed in the latest Solidity compiler, version 0.8.20, at the time of writing. Now, if encountered, the compiler provides the following informative error message: "Error: Derived contract must override function *validPurchase*. Two or more base classes define a function with the same name and parameter types.".

5.6 Improper Type Usage

This category groups vulnerabilities in which there is some misuse of types of data structures or functions.

 # 5.6.1 Missing return type on Function

This vulnerability refers to a missing return type in the definition of a smart contract interface. At runtime, if a contract that implements that interface contains two functions with the same name and arguments but have different return types, there is a chance that the wrong function will be called. This may lead to unexpected results once the calling contract receives the wrong data type (Zhang et al. 2019). This vulnerability is also known in the literature as *ERC 20 standard Violation* (Sunbeom et al. 2021), *ERC20 Interface* (Tsankov et al. 2018), or *missing return statement* (Hu et al. 2023).

This issue has been addressed in the latest Solidity compiler, version 0.8.20, at the time of writing. If encountered, the compiler provides the following informative error message: "Error: Overriding function return types differ".

 # 5.6.2 Function Return Type Mismatch

In this case, the developer implemented a function (starting from an interface), but it selected the wrong data type for the value to be returned (i.e., it differs from what is specified in the interface). This vulnerability is known in the literature in the context of non-fungible tokens by the name of *ERC721 Interface* (Tsankov et al. 2018) or *SWC-127 Arbitrary Jump with Function Type Variable* (SmartContractSecurity 2020).

 # 5.6.3 Parameter Type Mismatch

This issue refers to a divergence regarding the types of arguments used in a function that implements an interface. In this situation, even if the call is done with the right function name and arguments, the EVM considers it to be a non-existent function error. This vulnerability is also known in the literature as *ERC20 Indexed* (Tsankov et al. 2018) in the context of fungible tokens.

 # 5.6.4 Missing Type in Variable Declaration

In Solidity, the compiler infers the data type based on the assigned value whenever a variable is declared without an associated type. This additional computation may lead to higher costs (i.e., in gas) and memory usage and especially allows for overflow or underflow problems to occur. For instance, the compiler may infer that a signed integer is the right datatype for a certain variable, where an unsigned integer should be used. This vulnerability is also known in the literature as *unsafe type inference* (Tikhomirov et al. 2018)

This issue has been addressed in the latest Solidity compiler, version 0.8.20, at the time of writing. If encountered, the compiler provides the following informative error message: "Error: Expected primary expression."

 # 5.6.5 Wrong Type in Variable Declaration

This issue refers to a wrong selection of datatypes that leads to the allocation of more memory than what would be necessary for the intended function, leading to an increase in gas consumption. As an example, in Solidity, the `byte[]` type reserves 31 bytes of space for each element, whereas the `bytes` requires a single byte per element, thus being more space efficient. This vulnerability is also known in the literature as *byte array* (Tikhomirov et al. 2018), *global variable* (Li et al. 2022b), *type conversion errors* (Ding et al. 2021), or *unsafe array's length manipulation* (Shakya et al. 2022)

5.6.6 Wrong Type of Function

In Solidity, it is possible to specify a type for each function. Functions of type `view` can read data from state variables but cannot modify them, and no gas costs are involved, whereas functions of type `pure` neither can read nor modify state variables and similarly to view functions, no gas costs are associated with this type of function. This vulnerability occurs when a developer uses the wrong type for a function. For instance, there is an issue reported in Ethereum's GitHub (Ethereum's Github 2022) in which a state variable conversion operation (from storage to memory) inside a `pure` function results in a problem (i.e., to avoid this problem, the function type should be `view`). This vulnerability is also known in the literature as *unsafe type inference* (Tikhomirov et al. 2018).

This issue has been addressed in the latest Solidity compiler, version 0.8.20, at the time of writing. If encountered, the compiler provides the following informative warning message: "Warning: Function state mutability can be restricted to pure".

5.6.7 Non-Identifiable Order in Map Structure Iteration

In the Golang language, key-value pairs are not guaranteed to be unique when iterating through a Map structure. This potential lack of uniqueness can cause security issues, particularly if these uncertain values are present in operations that involve modifying the ledger. Such situations may lead to an inconsistent ledger state, which could compromise the ledger's integrity and reliability. This vulnerability is also known in the literature as *map structure iteration* (Li et al. 2022b).

5.7 Useless Code

This category groups a set of vulnerabilities in which the program contains a unit of code that, in practice, has no effect.

5.7.1 Unreachable Payable Function

This vulnerability refers to the case of contracts that allow the use of functions that accept credit but do not have any functionality for transacting it. They are insecure, as there is no way to recover the credit once it has been sent to the contract (Zhang et al. 2019). This vulnerability is also known in the literature as *be no black hole* (Chang et al. 2019), *disable transferring* (Ma et al. 2023), *freezing ether* (Choi et al. 2021; Ashraf et al. 2020; Jiang et al. 2018; Nguyen et al. 2020), *frozen ether* (Hu et al. 2023), *leaking ether to arbitrary address* (Hu et al. 2023), *locked ether* (Tsankov et al. 2018; Ashouri 2020; Mavridou et al. 2019), *locked funds* (Stephens et al. 2021), *locked money* (Shakya et al. 2022; Feist et al. 2019; Tikhomirov et al. 2018), *is_greedy* (Xing et al. 2020), *leaking ether* (Torres et al. 2021), *locked-ether* (Li et al. 2022d), *locking ether*(Torres et al. 2021), or *the ether lost or transferred* (Gupta et al. 2022)

This issue has been addressed in the latest Solidity compiler, version 0.8.20, at the time of writing. If encountered, the compiler provides the following informative warning message: "Warning: This function is named receive but is not the receive function of the contract. If you intend this to be a receive function, use receive(...) ... without the function keyword to define it".

5.7.2 No Effect Code Execution

This vulnerability refers to the presence of code that has no practical purpose (i.e., it has no effect on the intended functionality). Within a smart contract, it increases the size of the

program's binary code, which results in more gas consumption than would otherwise be necessary. This vulnerability is also known in the literature as *call to default constructor* (Tsankov et al. 2018), *dead code* (Chen et al. 2021), *useless assignment* (Hu et al. 2023), *code-no-effects* (Li et al. 2022d), or *SWC-135: Code With No Effects* (SmartContractSecurity 2020).

 # 5.7.3 Unused Variables

This vulnerability refers to the declaration of variables that are not used in the contract, which results directly in the allocation of unnecessary space in memory. As a consequence, the gas cost of executing the contract increases as well as the attack surface of the contract. Other effects are related to the readability or maintainability of the code. This vulnerability is also known in the literature as *redundant sstore* (Chen et al. 2021), *unused state variable* (Hu et al. 2023), *unused State Variable* (Tsankov et al. 2018), or *SWC-131 Presence of unused variables* (SmartContractSecurity 2020).

This issue has been addressed in the latest Solidity compiler, version 0.8.20, at the time of writing. If encountered, the compiler provides the following informative warning message: "Warning: Unused local variable. Warning: Unused function parameter. Remove or comment out the variable name to silence this warning".

 # 5.7.4 Inefficient Operation Sequence

Due to bad programming practices or outdated compilers (i.e., Solidity has more than 350 versions, with the first one being (Version 0.1.1) released in 2015), smart contracts may suffer from gas-inefficient operation sequences. Consequently, such contracts could be deployed with non-optimized bytecode, leading to increased resource and gas consumption. This vulnerability is also known in the literature as *SWAP1=DUP2=SWAP1* (Chen et al. 2021), *PUSHx=POP* (Chen et al. 2021), or *PUSH1=NOT* (Chen et al. 2021).

5.8 Version Issues

This category refers to issues that relate to the versioning of various aspects, including the use of deprecated versions of functions.

 # 5.8.1 Undetermined Program Version Prevalence

This vulnerability refers to the case where the developer allows a certain contract to be compiled across multiple versions. This allows the known faults in older versions to be easily activated. Zhang et al. (2019). This vulnerability is also known in the literature as *compiler version not fixed* (Tikhomirov et al. 2018), *Solc Version* (Tsankov et al. 2018), or *SWC-103: Floating Pragma* (SmartContractSecurity 2020).

 # 5.8.2 Outdated Compiler Version

Contracts that have been developed against an outdated compiler version can bring in several risks, mainly because newer versions may have resolved certain bugs or even introduced security mechanisms to avoid particular issues (e.g., the `throw` function has been disallowed in Solidity 0.5.0 and superior versions, in favor of `assert`, `require`/, and `revert`). This vulnerability is also known in the literature as *SWC-102: Outdated Compiler Version* (SmartContractSecurity 2020).

This issue has been addressed in the latest Solidity compiler, version 0.8.20, at the time of writing. If encountered, the compiler provides the following informative error message: "Error: Source file requires different compiler version (current compiler is 0.8.20+commit.a1b79de6.Darwin.appleclang) - note that nightly builds are considered to be strictly less than the released version".

 # 5.8.3 Use of Deprecated Functions

Deprecated functions are not recommended due to the fact that they are usually replaced by functions that solve known security issues or even operate in a more efficient manner (i.e., they may consume less gas). As an example, `sha3` was marked as a deprecated function in Solidity 0.5 and replaced by `keccak256`, which is more secure and efficient. This vulnerability is also known in the literature as *SWC-111: Use of deprecated solidity functions* (SmartContractSecurity 2020).

This issue has been addressed in the latest Solidity compiler, version 0.8.20, at the time of writing. If encountered, the compiler provides the following informative error message: "Error: suicide has been deprecated in favor of self-destruct".

5.9 Inadequate Data Representation

The numbers to represent the credits (e.g., Ether) can be very large (i.e., literals with many digits are difficult to read and review). Thus, it is recommended that the programmer use the native resources of the language to make this representation (e.g., Solidity 1000000000000000000 for 1 ether). This vulnerability is also known in the literature as *too-many-digits* (Li et al. 2022d; Tsankov et al. 2018).

5.10 Improper Modifier

This group gathers vulnerabilities that relate to the use of modifiers in functions and variables.

 # 5.10.1 Wrong Function Modifier

This vulnerability refers to the case of functions that are written solely to be used by other contracts (i.e., not within the contract). Such functions should be marked with the `external` modifier instead of `public`. The `public` modifier allows both external and internal calls. Marking a function with `external` results in gas savings, as every invocation will be using `calldata` (a special memory region to store arguments, which cannot be later modified by the function) and can avoid unnecessary read and write operations to memory, which occur with internal calls (i.e., that do not use `calldata`). This vulnerability is also known in the literature as *external-function* (Tsankov et al. 2018) or *SWC-100: Function Default Visibility* (SmartContractSecurity 2020).

 # 5.10.2 Missing Constant Modifier in Variable Declaration

Variables that are not modified during the execution flow should be declared as constants to save gas. In the absence of the `constant` modifier, it is assumed that the variable's value can be changed. This vulnerability is also known in the literature as *constable states* (Tsankov et al. 2018).

 # 5.10.3 Missing Visibility Modifier in Variable Declaration

Variables have different visibility states, which determine the context for accessing them. In Solidity, by default, the visibility of state variables and functions is `internal`, which allows access from functions in the same contract or derived contracts. A developer who is unaware of this may create a contract that allows exposure of sensitive data or allows unexpected behavior. This vulnerability is also known in the literature as *implicit Visibility* (Ashizawa et al. 2021) , *state variables default visibility* (Tsankov et al. 2018), *visibility level* (Tikhomirov et al. 2018), or *SWC-108: State Variable Default Visibility* (SmartContractSecurity 2020).

5.11 Redundant Functionality

Contracts that are written with redundant functionality increase code size and make maintainability difficult. In a simple scenario, a programmer creates a function and later (by bad practices) ends up creating the same functionality again in a new function. He/she identifies a vulnerability in the new function and fixes it, but the old function with the vulnerability is used by the caller. This vulnerability is also known in the literature as *redundant fallback function* (Shakya et al. 2022) or *redundant fallback function* (Tikhomirov et al. 2018).

5.12 Shadowing

This category groups vulnerabilities in which there are code elements (e.g., a function or a variable) with the same name, which can lead to erroneous and unexpected behavior.

 # 5.12.1 Use of Same Variable or Function Name In Inherited Contract

When using the same name as a local variable, which was previously declared by an inherited contract, the program loses the reference of the inherited variable, causing the local variable to assume the role of the other variable. This vulnerability is also known in the literature as *shadow memory* (Ashouri 2020), *shadowing state variables* (Tsankov et al. 2018), *shadowing* (Feist et al. 2019), or *SWC-119: Shadowing State Variables* (SmartContractSecurity 2020).

This issue has been addressed in the latest Solidity compiler, version 0.8.20, at the time of writing. If encountered, the compiler provides the following informative error message: "Error: Identifier already declared".

 # 5.12.2 Variables or Functions Named After Reserved Words

This bug occurs when creating variables named after keywords of the language itself. For example, in Solidity, creating a variable with the name *now* conflicts with the function that returns the date and time. This vulnerability is also known in the literature as *shadowed builtin* (Tsankov et al. 2018) or *shadowing-builtin* (Li et al. 2022d).

This issue has been addressed in the latest Solidity compiler, version 0.8.20, at the time of writing. If encountered, the compiler provides the following informative warning message: "Warning: This declaration shadows a built-in symbol".

 # 5.12.3 Use of the Same Variable or Function Name In a Single Contract

This vulnerability refers to cases where the same name is used for more than one variable or function inside the contract. This makes the program lose the reference of the variable of the class, assuming the variable of the function as its role. This vulnerability is also known in the literature as *redefined variable* (Hu et al. 2023) or *shadowed local variable* (Tsankov et al. 2018).

This issue has been addressed in the latest Solidity compiler, version 0.8.20, at the time of writing. If encountered, the compiler provides the following informative warning message: "Warning: This declaration shadows an existing declaration".

5.13 Buffer-Based Vulnerabilities

This category refers to buffer-related vulnerabilities (e.g., stack-based, heap-based, bugger over-read) in which it is possible to write more data than what the buffer can hold, thus modifying memory areas outside the expected or read the buffer using mechanisms such as indexes or pointers that reference memory locations after the targeted buffer.

 # 5.13.1 Stack-based Buffer Overflow

The EVM keeps an execution stack that manages the execution of contracts. If an attacker is allowed to overflow this stack (e.g., by using specially crafted inputs), it can potentially overwrite control variables (e.g., timestamp or block number) and, for instance, gain unauthorized access to certain resources. This vulnerability is also known in the literature as *Stack size limited* (Atzei et al. 2017).

 # 5.13.2 Write to Arbitrary Storage Location

In solidity, arrays are stored as contiguous fixed-size slots. In the absence of a bounds verification, a malicious user could write data to a particular storage slot used to store the contract owner's address, which could be overwritten and then used to harm the contract further. This vulnerability is also known in the literature as *arbitrary write* (Choi et al. 2021), *buffer-overwrite* (Pani et al. 2023), *storage modification* (Krupp and Rossow 2018), *unrestricted write* (Tsankov et al. 2018), or *SWC-124: Write to Arbitrary Storage Location* (SmartContractSecurity 2020).

 # 5.13.3 Read from Arbitrary Storage Location

This vulnerability is related to the incorrect manipulation of data in a buffer, which usually causes buffer underflow or buffer overflow. A buffer underflow occurs when a program tries to read, write, or update data in a buffer (such as an array) before the buffer's starting point. In other words, it attempts to access an index that lies before the allocated memory to the buffer. In contrast, a buffer overflow occurs when a program tries to access an index after the allocated memory to the buffer. This vulnerability is referred to in the literature as *Buffer-overflow* (Pani et al. 2023) or *Buffer-underflow* (Pani et al. 2023).

5.14 Use of Malicious Libraries

This vulnerability refers to the use of third-party libraries containing malicious code. This vulnerability is also known in the literature as *external library calling* (Li et al. 2022b) or *malicious libraries* (Tikhomirov et al. 2018).

5.15 Typographical Error

This vulnerability refers to single-digit errors made by programmers while typing source code, e.g., in logic or arithmetic operations. For example, for value assignment, a developer may type by mistake "+ =" instead of "=" or may use "−" instead of "+" or "−−" instead of "++" (Hartel and Schumi 2020). This vulnerability is also known in the literature as *SWC-129: Typographical Error* (SmartContractSecurity 2020).

This issue has been addressed in the latest Solidity compiler, version 0.8.20, at the time of writing. If encountered, the compiler provides the following error message: "Error: Use of unary + is disallowed".

5.16 Wrong Logic

This vulnerability refers to when the developer makes a systematic logic mistake in the contract, leading repeatable executions to unexpected results. This vulnerability is also known in the literature as *Logic loopholes* (Ding et al. 2021).

5.17 Wrong Moment for Token Generation

This vulnerability arises when the token generation process within the contract relies solely on the validity of the receiving address. As a result, malicious programmers can exploit this weakness to generate tokens arbitrarily, gaining the ability to manipulate the token price. This vulnerability is also known in the literature as *Generate Token After ICO* (Ma et al. 2023).

6 Incorrect Control Flow

This category groups a set of vulnerabilities that, if exploited, cause changes in the control flow of the program.

6.1 Incorrect Sequencing of Behavior

This category gathers vulnerabilities that end up in a sequence of behaviors that are carried out in the wrong order, leading to unexpected results.

 # 6.1.1 Incorrect Use of Event Blockchain Variables for Time

Contracts that rely on using block control information (i.e., timestamp, coinbase, number, difficulty, and gas limit) for sequential event control are vulnerable to tampering by the miner. This vulnerability is also known in the literature as *block info dependency* (Chen et al. 2022), *block information dependency* (Li et al. 2022c), *block no. dependency* (Ashraf et al. 2020; Jiang et al. 2018), *block number dependency* (Chen et al. 2020b; Nguyen et al. 2020), *block state dependence* (Kalra et al. 2018), *block state dependency* (Choi et al. 2021), *block Timestamp* (Zhang et al. 2022a), *event-ordering (EO) bugs* (Kolluri et al. 2019), *race condition* (Ashouri 2020), *system timestamp* (Li et al. 2022b), *time dependency* (Ashizawa et al. 2021), *time dependence* (Yu et al. 2021), *time dep* (Zhang et al. 2022a), *timestamp dependency* (Chen et al. 2020b; Hwang et al. 2022; Hu et al. 2023; Ashraf et al. 2020; Song et al. 2019; Ma et al. 2022; Akca et al. 2019; Jiang et al. 2018; Nguyen et al. 2020; Wang et al. 2021), *timestamp dependence* (Liu et al. 2021; Zhuang et al. 2020; Luu et al. 2016; Tikhomirov et al. 2018), *timestamp manipulation* (Liao et al. 2022), *timestamp* (Sun et al. 2023; Zeng et al. 2022; Tsankov et al. 2018), *block dependency* (Torres et al. 2021), *timestamp expansion* (Gupta et al. 2022), *timestamp* (Li et al. 2022d), or *SWC-116: Block values as a proxy for time* (SmartContractSecurity 2020).

 # 6.1.2 Incorrect Function Call Order

This vulnerability refers to the creation of public functions that expect to be called in a certain sequence, originating unanticipated results whenever clients do not follow the right call order (Mavridou and Laszka 2018). This vulnerability is also known in the literature as *concurrency of program* (Li et al. 2022b), *the transaction ordering* (Gupta et al. 2022), *transaction ordering dependence (TOD)* (Zeng et al. 2022), *transaction order dependency* (Torres et al. 2021), or *SWC-114: Transaction Order Dependence* (SmartContractSecurity 2020).

 # 6.1.3 Improper Locking

This issue refers to the case where a contract assumes that all entities participating in a transaction must have the same credit balance before the contract operations can execute. If there are no adequate (e.g., wrong or even missing) locking mechanisms, an attacker can forcefully send credit to the other entity, which would cause the verification of the balance

condition never to be met. Thus, the contract may become unusable or show unexpected behavior (or unexpected state changes). This vulnerability is also known in the literature as *arbitrary sending of ether* (Feist et al. 2019), *balance equality* (Tikhomirov et al. 2018), *checking for strict balance equality* (Shakya et al. 2022) , *incorrect Equality* (Tsankov et al. 2018), *strict check for balance* (Chen et al. 2020b), or *SWC-132: Unexpected Ether balance* (SmartContractSecurity 2020).

 # 6.1.4 Transfer Pre-Condition Dependent on Transaction Order

In the case of this vulnerability, the order in which transactions are executed influences a pre-condition that guards the execution of the transfer. This influence may erroneously result in, for instance, a transaction not being executed at all. This vulnerability is known in the literature as *front running* (Crincoli et al. 2022), *TOD Transfer* (Tsankov et al. 2018), *TOD* (Sun et al. 2023; Zhang et al. 2022a; Fu et al. 2019; Bose et al. 2022), or *SWC-114: Transaction Order Dependence* (SmartContractSecurity 2020).

 # 6.1.5 Transfer Amount Dependent on Transaction Order

This issue refers to the case where the value of the variable that stores or determines an amount of a digital asset (to be transferred) is modified before it is sent to the recipient due to transaction ordering within a block. The amount may be changed due to the effect of multiple transactions being grouped in a block and executed in a specific order, producing unexpected changes in the value being transferred. This vulnerability is also known in the literature as *TOD Amount* (Tsankov et al. 2018), *TOD* (Bose et al. 2022; Wang et al. 2021), or *SWC-114: Transaction Order Dependence* (SmartContractSecurity 2020).

 # 6.1.6 Transfer Recipient Dependent on Transaction Order

In the case of this vulnerability, the transfer recipient is modified before the send event due to transaction ordering within a block. For example, if the intended recipient address is stored as a storage variable and a transfer is to execute based on this address, there is a chance the address may be changed or overwritten by another transaction before the transfer. This vulnerability is also known in the literature as *direct value transfer* (Krupp and Rossow 2018), *TOD Receiver* (Tsankov et al. 2018), *TOD* (Bose et al. 2022), *transaction order dependence* (Kalra et al. 2018), *transaction order dependency* (Hu et al. 2023), *transaction-ordering dependence* (Luu et al. 2016), *transaction-ordering dependence* (Song et al. 2019), or *SWC-114: Transaction Order Dependence* (SmartContractSecurity 2020).

 # 6.1.7 Exposed State Variables

This vulnerability refers to the case where a developer erroneously exposes a state variable, whose value may then be modified by an attacker so that this modification influences the execution of a certain contract operation. As an example, consider a contract that executes a credit transfer from one user to another and has a `require` statement for verifying that there is sufficient credit to conclude the operation. If the balance is stored as a public state variable, a malicious use could change its value so that the `require` is avoided, allowing the user to run a transfer that exceeds the amount of credit the malicious user holds. This vulnerability is also known in the literature as *reified object addresses* (Li et al. 2022b) or *vulnerable state* (Krupp and Rossow 2018).

 # 6.1.8 Wrong Definition of Actions

EOSIO facilitates inter-contract communication through the use of inline actions and defer actions. When a smart contract invokes another contract using inline actions, all these actions are bundled together into a single transaction under the control of the caller. In contrast, defer actions will be executed in a separate transaction and cannot be reverted by the caller. In certain situations, attackers might exploit the ability to rollback inline actions, manipulating the blockchain to deny specific transactions. To mitigate such risks, developers can strategically implement defer actions to create a more secure contract execution flow, protecting against rollback attacks. This vulnerability is also known in the literature as *rollback* (Chen et al. 2022).

 # 6.2 Inadequate Input Validation

This group refers to vulnerabilities involving the inadequate validation of functional conditions, which are requirements that a contract must meet so that it can operate correctly. Such conditions may offer protection against certain types of attacks or force certain business rules to be followed.

 # 6.2.1 Improper Input Validation

This type of problem occurs when an attacker calls a certain contract operation using invalid or malicious input data, capable of affecting the functioning of the contract due to the fact that either it does not validate the incoming inputs or validates them in an incorrect manner. For instance, in the context of Solidity, a Short Address Attack occurs when a contract receives fewer data than expected, leading the system to fill the missing bytes with zeros (Chen et al. 2020b). As a consequence, the behavior may become unexpected if the code assumes that the input data will comply with a certain length or format. This vulnerability is also known in the literature as *avoid non-existing address* (Chang et al. 2019), *invalid input data* (Chen et al. 2020b), *short address attack* (Ashouri 2020; Crincoli et al. 2022), *shortening of address* (Mavridou et al. 2019), *unchecked input arguments* (Li et al. 2022b), *unchecked tainted static call* (Brent et al. 2020), *has_short_address* (Xing et al. 2020), or *shift-parameter-mixup* (Li et al. 2022d)

 # 6.2.2 Extraneous Input Validation

In this particular case, the functional conditions of the contract are too strong and do not allow certain behaviors (which would be valid) to occur, making the contract unable to meet the requirements. This vulnerability is also known in the literature as *requirement violation* (Choi et al. 2021) or *SWC-123: Requirement violation* (SmartContractSecurity 2020).

 # 7. Arithmetic Issues

This category groups different vulnerabilities that share the outcome of resulting in arithmetic problems.

 # 7.1 Overflow and Underflow

This category refers to the use of operations (e.g., addition, subtraction) over values that result in a value that is less than (or greater than) the minimum values (or maximum value) that a variable can hold, which produces a value different from the correct result.

 # 7.1.1 Integer Underflow

This vulnerability refers to operations over an Integer variable that results in a value that is less than the minimum value allowed by the Integer type. This vulnerability is also known in the literature as *arithmetic bugs* (Torres et al. 2018), *flow* (Sun et al. 2023), *integer underflow vulnerability* (Song et al. 2019), *integer bug* (Choi et al. 2021), *integer over/underflow* (Sunbeom et al. 2021; Wang et al. 2019; So et al. 2020; Kalra et al. 2018), *integer underflow* (Ashizawa et al. 2021; Jin et al. 2023; Kalra et al. 2018; Stephens et al. 2021; Wang et al. 2021), *integer overflow and integer underflow* (Gupta et al. 2022; Ayoade et al. 2019; Ashouri 2020; Nguyen et al. 2020), *overflow/underflow* (Cui et al. 2022; Ma et al. 2022; Akca et al. 2019), *overflow and underflow* (Hu et al. 2023), *underflow* (Zhang et al. 2022a; Zhou et al. 2022b), *has_flows* (Xing et al. 2020), *multi-transaction sequence vulnerabilities* (Zhang et al. 2022b), or *SWC-101: Integer Overflow and Underflow* (SmartContractSecurity 2020).

 # 7.1.2 Integer Overflow

This vulnerability refers to operations over an Integer variable that results in a value that is larger than the maximum value allowed by the Integer type. This vulnerability is also known in the literature as *arithmetic Bugs* (Torres et al. 2018), *flow* (Sun et al. 2023), *integer bug* (Choi et al. 2021), *integer over/underflow* (Sunbeom et al. 2021; Wang et al. 2019), *integer overflow vulnerability* (Song et al. 2019), *integer overflow* (Grech et al. 2020; Fu et al. 2019; Ding et al. 2021; Ashizawa et al. 2021; Jin et al. 2023; Zhou et al. 2022b; Stephens et al. 2021; Wang et al. 2021), *integer overflow and integer underflow* (Gupta et al. 2022; Hu et al. 2023; Ayoade et al. 2019; Ashouri 2020; Nguyen et al. 2020; So et al. 2020), *underflow* (Cui et al. 2022), *overflow/underflow* (Akca et al. 2019; Ma et al. 2022), *overflow detector* (Gao et al. 2019), *overflow* (Zhang et al. 2022a; Torres et al. 2021), *has_flows* (Xing et al. 2020), or *SWC-101: Integer Overflow and Underflow* (SmartContractSecurity 2020).

 # 7.2 Division Bugs

This category groups issues related to erroneous division operations.

 # 7.2.1 Divide by Zero

This issue refers to the attempt of a program to divide a value by zero. This vulnerability is also known in the literature as *arithmetic bugs* (Torres et al. 2018), *divide by zero* (Sunbeom et al. 2021), *divide-by-zero* (Pani et al. 2023), *division by zero* (Akca et al. 2019), *division-by-zero* (So et al. 2020), or *zero division risk* (Gupta et al. 2022).

 # 7.2.2 Integer Division

At the time of writing, a smart contract mainstream language like Solidity does not support floating point or decimal types. Thus, the remainder of a division operation is always lost. Developers may use fixed-point arithmetic and external libraries to handle this kind of operation. This vulnerability is also known in the literature as *integer division* (Tikhomirov et al. 2018), *numerical precision error* (Cui et al. 2022), or *SWC-101: Integer Overflow and Underflow* (SmartContractSecurity 2020).

 # 7.3 Conversion Bugs

This category groups a set of vulnerabilities where there are issues related to the conversion between different datatypes.

 # 7.3.1 Truncation Bugs

This vulnerability refers to the case where a variable declared in a certain type is converted to a smaller type, which means that data is lost during the conversion process. This vulnerability is also known in the literature as *truncation bugs* (Torres et al. 2018) or *SWC-101: Integer Overflow and Underflow* (SmartContractSecurity 2020).

 # 7.3.2 Signedness Bugs

The conversion of a signed integer type to an unsigned type of the same width may change a negative value to a positive one (the opposite may also happen) (Torres et al. 2018). This vulnerability is also known in the literature as *Signedness bugs* (Torres et al. 2018) or *SWC-101: Integer Overflow and Underflow* (SmartContractSecurity 2020).

8 Improper Access Control

This category groups a set of vulnerabilities that are strongly related to authentication or access control.

8.1 Incorrect Authentication or Authorization

The smart contract fails to identify a client or determine its privileges properly, resulting in wrong access privileges for that particular client.

 # 8.1.1 Wrong Caller Identification

In Solidity, `tx.origin` allows obtaining the address of the account that initiated a transaction and `msg.sender` allows obtaining the address of the contract that has called the function being executed. The use of the `tx.origin` for access control may be a way of opening an entry point to a malicious user. A malicious user may create a contract that calls the vulnerable function (i.e., the one that uses `tx.origin` to check the identity of the caller). Thus, `msg.sender` will differ from `tx.origin`. In the case the vulnerable function uses `tx.origin` for access control, it will allow the user to perform actions it should not be able to. This vulnerability is also known in the literature as *incorrect check for authorization* (Chen et al. 2020b), *missing authorization verification* (Chen et al. 2022), *missing access control checks (MACC)* (Ghaleb et al. 2023), *the abuse of tx.origin* (Ye et al. 2022), *transaction origin use* (Choi et al. 2021), *transaction state dependence* (Kalra et al. 2018), *tx-origin* (Xue et al. 2022; Li et al. 2022d; Hwang et al. 2022; Gupta et al. 2022; Tikhomirov et al. 2018; Sun et al. 2023; Hu et al. 2023; Zhang et al. 2022a; Akca et al. 2019; Tsankov et al. 2018), *use of tx.origin* (Zeng et al. 2022), *using tx.origin for authorization* (Shakya et al. 2022), or *SWC-115: Authorization through tx.origin* (SmartContractSecurity 2020).

 # 8.1.2 Owner Manipulation

This vulnerability allows an attacker to exploit some function or feature of the smart contract by manipulating the owner control variable. This allows the attacker to perform some kind of restricted operations (Zhang et al. 2020b). This vulnerability is also known in the literature as *freeze account* (Ma et al. 2023), *missing owner check* (Cui et al. 2022), *taint for owner* (Liao et al. 2022), *tainted owner variable* (Brent et al. 2020), *unprotected function* (Stephens et al. 2021), or *vulnerable access control* (Zhang et al. 2020b).

 # 8.1.3 Missing Verification for Program Termination

This issue refers to the lack of a secure verification for terminating a published (deployed) contract, allowing an attacker to terminate it in an unauthorized manner. `Selfdestruct` is an EVM instruction that is able to nullify the bytecode of a deployed contract. When invoked, it stops the execution of the EVM, deletes the contract's bytecode, and sends the remaining funds to a certain address. Access to this kind of function by non-authorized clients may result in security issues. This vulnerability is also known in the literature as *accessible self-destruct* (Brent et al. 2020), *destroy token* (Ma et al. 2023), *guard suicide* (Chang et al. 2019), *self-destruct abusing* (Ye et al. 2022), *self-destruct* (Zeng et al. 2022), *suicidal contract* (Choi et al. 2021), *suicidal contracts* (Feist et al. 2019), *tainted self-destruct* (Brent et al. 2020), *unprotected suicide* (Hu et al. 2023; Mavridou et al. 2019), *unrestricted Self-destruct* (Tsankov et al. 2018), *multi-transaction sequence vulnerabilities* (Zhang et al. 2022b), *suicide* (Fu et al. 2019), *unprotected self-destruct* (Torres et al. 2021), or *SWC-106: Unprotected SELFDESTRUCT Instruction* (SmartContractSecurity 2020).

8.2 Improper Protection of Sensitive Data

This category generally refers to the issues that result in the inability to protect sensitive information from unauthorized clients.

 # 8.2.1 Exposed Private Data

This issue refers to the cases in which contracts store unencrypted sensitive data in public blockchain transactions. Solidity, like other programming languages, supports the `private` keyword that indicates that data is only accessible within the contract itself. However, in blockchain environments, marking a variable with `private` does not make it fully invisible to the outside world. Miners, who are responsible for validating transactions on the blockchain, can view the code of the contract and the value of its state variables (Zhang et al. 2019). This vulnerability is also known in the literature as *exposed secret* (Zhang et al. 2020b), *private modifier* (Tikhomirov et al. 2018), *private-not-hidedata* (Li et al. 2022d), or *SWC-136: Unencrypted Private Data On-Chain* (SmartContractSecurity 2020).

 # 8.2.2 Dependency on External State Data (Unsolvable Constraints of External Critical State Data)

This vulnerability refers to the use of data that is not under control nor is generated by the contract (i.e., external critical state data). A malicious user may exploit this situation if such data determines the outcome of the execution of the contract. This vulnerability is also known in the literature as *Unsolvable constraints* (Zhang et al. 2020b).

8.3 Cryptography Misuse

This category groups vulnerabilities that generally reflect misuse of cryptography mechanisms.

 # 8.3.1 Incorrect Verification of Cryptographic Signature

This issue refers to the wrong verification of the authenticity and integrity of messages with the use of message signatures. As an example, a developer could develop a vulnerable contract that relies on a signature in a signed message hash for representing the earlier verification of previous messages. A client could generate a malicious message with a valid signature and include it in the hash. The contract then would validate the signature and update the hash,

indicating that the message was processed. This vulnerability is also known in the literature as *check signature* (Nishida et al. 2021), *missing key check* (Cui et al. 2022), *signature-malleability* (Li et al. 2022d), or *SWC-117: Signature Malleability* (SmartContractSecurity 2020).

 # 8.3.2 Improper Check against Signature Replay Attacks

This vulnerability refers to a situation where a malicious client is able to obtain the message hash of a legitimate transaction and is allowed to use the same signature to impersonate the legitimate client and execute fraudulent transactions. This vulnerability is also known in the literature as *SWC-121: Missing Protection against Signature Replay Attacks* (SmartContract-Security 2020).

 # 8.3.3 Improper Authenticity Check

In this case, a contract may tolerate off-chain signed messages instead of waiting for an on-chain signature. This is usually done with the goal of improving performance but may come at the expense of compromising the authenticity of the message. This vulnerability is also known in the literature as *Missing Signer Check* (Cui et al. 2022), *SWC-122: Lack of proper signature verification* (SmartContractSecurity 2020).

 # 8.3.4 Incorrect Argument Encoding

This vulnerability refers to the misuse of one-way hash functions (i.e., Solidity keccak256) namely in the incorrect encoding of the function arguments, which can result in a higher likelihood of hash collisions for different entries. This vulnerability is also known in the literature as *Authorization* (Mavridou and Laszka 2018), *Hash collision* (Lu et al. 2019) or *SWC-133: Hash Collisions With Multiple Variable Length Arguments* (SmartContractSecurity 2020).

5 Discussion

This section overviews the main characteristics of the taxonomy and maps our observations to state-of-the-art and industry practices. This section also provides a brief summary of the main aspects that contribute to the overall quality of the taxonomy. Additionally, we conduct a comparative analysis between our taxonomy and existing community-based classifications. Lastly, we detail the process followed for the expert-based validation and present the obtained results.

5.1 Mapping the Taxonomy to State of the Art

Table 14 summarizes the distribution of the number of vulnerabilities per each of the main categories present in our taxonomy. As we can see, the distribution is dominated by *Bad Programming Practices & Language Weaknesses*, which account for almost half of the vulnerabilities. The majority of the remaining vulnerabilities exhibit a relatively uniform distribution.

Figure 6 further characterizes the identified vulnerabilities, namely by identifying the different 'defect types' (in the y-axis) and specifying the number of OpenSCV vulnerabilities per each 'defect type' (between parenthesis, in the y-axis). The plot then shows the prevalence of the 'qualifier' values (i.e., *missing*, *wrong*, and *extraneous*). Notice that the sum of the

Table 14 Vulnerability distribution across OpenSCV's main categories

Level	Qtd defects
Unsafe external calls	14
Mishandled events	5
Gas depletion	2
Erroneous credit transfer	6
Bad programming practice & language weakness	42
Incorrect control flow	10
Arithmetic issues	6
Improper access control	9

qualifier values exceeds the vulnerability count (between parenthesis in the y-axis), as a certain vulnerability may be associated with more than one qualifier (e.g., vulnerability may occur due to a *missing checking* or a *wrong checking*).

As we can see in Fig. 6, the top 'defect types' belong to *Assignment/Initialization*, *Interface/O-O Messages*, and *Checking*, which is closely followed by *Algorithm/Method*. The top three 'defect types' account for nearly two-thirds of the vulnerabilities, and the top four account for more than 80% of the 94 vulnerabilities.

Table 15 summarizes the qualifiers' distribution. It also shows the distribution of the combined qualifiers (e.g., *wrong* and *missing* together), which represent vulnerabilities whose correction may be related to more than one qualifier (e.g., a certain vulnerability may be due to a *missing* or due to a *wrong assignment*). As shown, the *wrong* qualifier is the most frequent one, followed by *missing*. In terms of combinations, *missing* and *wrong* together are the most frequent cases.

We selected three different cases of classification schemes for comparison with OpenSCV. In particular, we selected SWC (SmartContractSecurity 2020) for frequently appearing in the literature, we also selected the classification presented by Rameder et al. (2022) for being the most extensive one found in the literature, and we also selected the list of vulnerabilities presented in Hu et al. (2023) for being the most recent vulnerability detection work. Figure 7 shows to what extent these classifications map the vulnerabilities identified in our taxonomy.

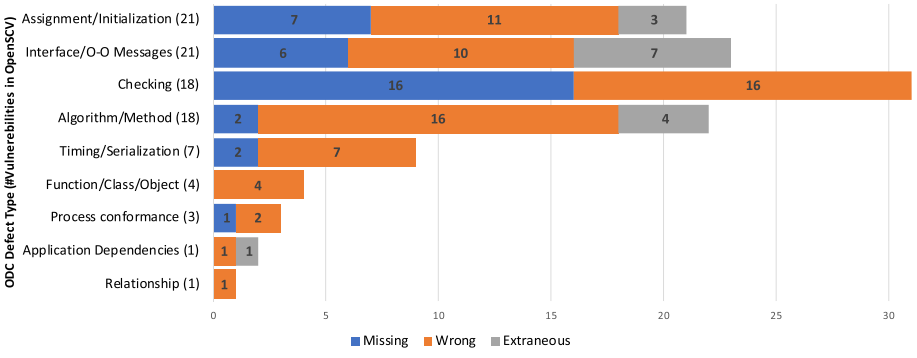

Fig. 6 Vulnerabilities per ODC defect type and qualifier

Table 15 ODC qualifier distribution

Qualifier	Extraneous	Missing	Wrong
Extraneous	16	1	3
Missing	1	34	21
Wrong	3	21	68

The first observation from Fig. 7 reveals that OpenSCV currently includes a larger number of distinct vulnerabilities compared to the remaining classifications. While SWC includes 37 vulnerabilities, our detailed analysis has revealed that we can map these to 49 different vulnerabilities documented in OpenSCV. In contrast, among the 54 defects (not all of which are vulnerabilities) presented in Rameder et al. (2022), only 45 have been successfully mapped to vulnerabilities in OpenSCV. Furthermore, all 18 vulnerabilities presented in Hu et al. (2023) are mapped with the 18 vulnerabilities in OpenSCV.

We also compared OpenSCV with existing community-oriented classification schemes. Table 16 presents this comparison. We can observe that OpenSCV presents a greater coverage (i.e., 94 vulnerabilities) and holds important characteristics compared to other classifications. OpenSCV not only provides references to external classifications it also provides links to the vulnerability detection tools. It also provides both vulnerable and fixed codes, like most classifications. However, in contrast to other classifications, it specifies the compiler version in which the vulnerability is fixed. Moreover, it follows a tree structure, which is more fine-grained and flexible to updates. At the time of writing and to the best of our knowledge, OpenSCV is the most up-to-date vulnerability classification scheme with coverage for several blockchain platforms.

In comparison to the two of the most notable classifications, namely DASP and SWC, OpenSCV offers distinct advantages. Both DASP and SWC classifications lag behind the current state of the practice. The authors of Ivanov et al. (2023) highlight a significant challenge in vulnerability mapping using SWC and DASP classifications. They point out that the inherent difficulty in mapping vulnerabilities with the SWC is due to discrepancies in the nomenclature used by various security tools. This problem arises because both the DASP and SWC classifications do not incorporate the vulnerability lists provided by the vulnerability detection tools. In contrast, OpenSCV presents a solution to this problem. OpenSCV helps

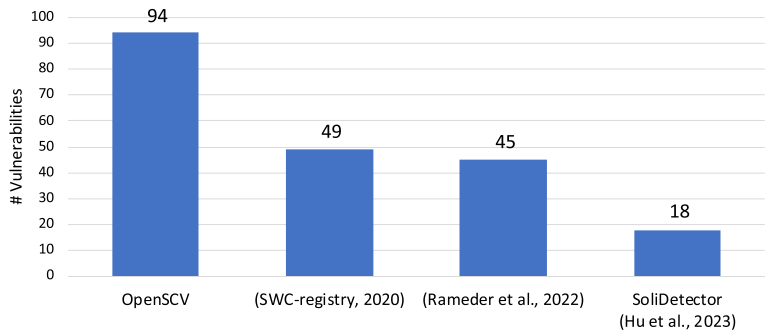

Fig. 7 Mapping of the identified vulnerabilities to other classifications

Table 16 Comparison with community-based classification schemes

Classifications	# Vuln.	Link to external classification	Link to detection tools	Code example	Structure	Open source	Compiler version specified	Last update	Blockchain plataforms
SWC	37	Y	N	Vulnerable & Fixed Codes	List	Y	N	2018	Ethereum
DASP	9	N	N	Vulnerable Codes	List	Y	N	2018	Ethereum
SIGP	16	N	N	Vulnerable & Fixed Codes	Group	Y	N	2018	Ethereum
SMARTDEC	11	Y	N	–	Group	Y	N	2018	Ethereum
OpenSCV	94	Y	Y	Vulnerable & Fixed Codes	Tree	Y	Y	2023	Ethereum Hyperledger EOS Tezos

developers seamlessly bridge the gap between the vulnerability names reported by different tools and their corresponding entries in the SWC classification.

The authors of Ivanov et al. (2023) also highlighted another weakness of these classifications (i.e., DASP, SWC), which is the discrepancy between them and the vulnerability list detected by the vulnerability detection tools. To illustrate this point, consider the case of *Reentrancy* vulnerabilities, which can be detected by tools like Slither and Securify. These tools distinguish between different types of reentrancy issues, which may include problems leading to financial leaks or those affecting the overall system state. SWC and DASP, however, tend to provide a broader, less detailed classification. Another example is related to the variable initialization vulnerabilities. In SWC-109, there is a specific entry to handle storage variable issues. However, several tools report different kinds of problems related to variable initialization, such as *uninitialized variables* (Feist et al. 2019), *uninitialized-local* (Tsankov et al. 2018). Regarding this aspect, OpenSCV fits better because it places both issues under the category of *Improper Declaration or Initialization* and specifies each of them with two distinct entries: *5.2.3 Missing Variable Initialization* and *5.2.4 Uninitialized Storage Variables*.

Table 17 presents a list of the 10 vulnerabilities most frequently discussed in the works analyzed in this paper. The names in Table 17 have been normalized to use the OpenSCV vulnerability names. The 'index' column is a reference to the OpenSCV vulnerability, the 'vulnerability' column holds the name of the vulnerability in our taxonomy, then we show to which SWC or DASP names it maps, and finally the 'count' column indicates the number of works that discuss that particular vulnerability, albeit using different names. As we can see in Table 17 two of the top vulnerabilities (i.e., *Unreachable Payable Function* and *Improper Use of Exception Handling Functions*) do not map to any vulnerability in SWC. Similarly, three of the vulnerabilities do not have a corresponding one in DASP, namely the two aforementioned cases plus *Missing verification for program termination*.

We also made a comparison between OpenSCV and the state of the practice (in terms of tools) by analyzing and mapping their announced vulnerability detection capabilities and the vulnerabilities of our taxonomy.

Table 17 The top 10 prevalent vulnerabilities identified in vulnerability detection research

Index	Vulnerability	SWC	DASP	Count
1.1.1	Unsafe credit transfer	SWC-107	DASP-1	61
6.1.1	Incorrect use of event blockchain variables for time	SWC-116	DASP-8	43
7.1.2	Integer overflow	SWC-101	DASP-3	32
7.1.1	Integer underflow	SWC-101	DASP-3	30
5.7.1	Unreachable payable function	–	–	26
8.1.1	Wrong caller identification	SWC-115	DASP-2	24
8.1.3	Missing verification for program termination	SWC-106	–	18
2.1.2	Improper exception handling in a loop	SWC-128	DASP-1	15
2.1.1	Improper use of exception handling functions	–	–	15
1.6	Delegatecall to untrusted Callee	SWC-112	DASP-2	15

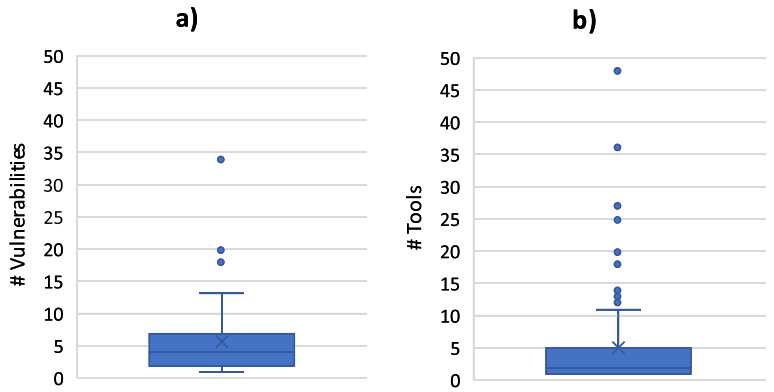

Fig. 8 Announced detection capabilities of current tools: a) the average number of vulnerabilities detected per tool; b) the average number of tools per vulnerability

Figure 8.a) shows the practical distance of the 77 identified tools/works in vulnerability detection to our current state of knowledge in what concerns smart contract vulnerabilities. As we can see in Fig. 8.a), the existing tools are able to detect on average 5.5 of the vulnerabilities presented in OpenSCV. Most tools are detecting from 2 to 7 different vulnerabilities. The best vulnerability detection tools (in terms of detection capabilities) are: Securify (Tsankov 2018), covering 36% of OpenSCV vulnerabilities (34 out of 94); Smartcheck (Tikhomirov et al. 2018) with 21%, (20 out of 94) and; SoliDetector (Hu et al. 2023) and HFCCT which detect 20% of the vulnerabilities of OpenSCV (18 out of 94). These results indicate that there is significant potential for improvement in vulnerability detection, and combining the diverse capabilities of various tools could be a promising approach for the development of a more effective solution.

Figure 8.b) shows, from the perspective of each individual vulnerability, how many tools are able to detect it. As shown, on average, each vulnerability is detected by 5.14 tools (most of them are detected by 1 and 5 tools).

Figure 9 shows the focus of the different classes of tools (i.e., Formal Methods, Static Code Analysis, Software Testing, and Machine Learning) per each of the top categories in our taxonomy. As shown in the figure, the category *1. Unsafe External Calls* has the largest

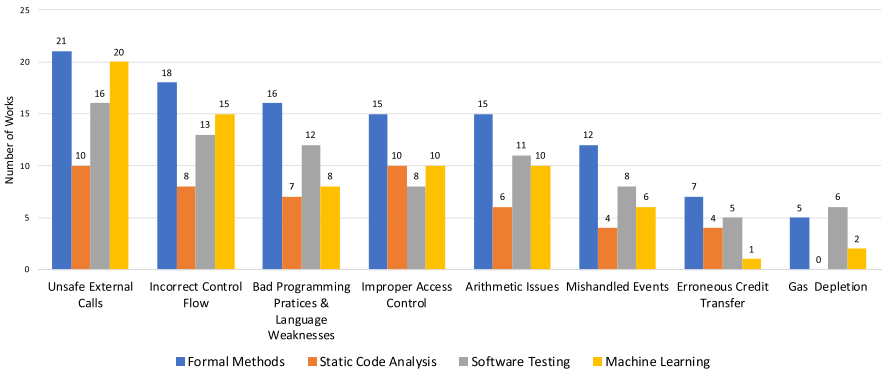

Fig. 9 Announced detection capabilities for each of the taxonomy top categories

focus on vulnerability detection tools and is mostly dominated by formal methods, despite the recent growth of tools based on machine learning. We also observe that although, *5. Bad Programming Practices & Language Weaknesses* is the category with the largest set of vulnerabilities (i.e., 42 OpenSCV vulnerabilities), proportionally it is not gathering the same attention as other categories (e.g., *6. Arithmetic Issues* is being targeted by 40 tools, although it includes only 6 OpenSCV vulnerabilities). Furthermore, we observe that tools based on Formal Methods are the most prevalent across all categories of vulnerabilities, with the exception of *3. Gas Depletion*, where software testing tools slightly outnumber the former.

5.2 Validation of OpenSCV

To validate OpenSCV, we performed an expert-based analysis. To do so, we conducted a questionnaire regarding the overall quality of our taxonomy. We invited experts (researchers) who are/were active in smart contract security and requested their opinions about the vulnerabilities and the categories defined in OpenSCV. As we intend to bring the OpenSCV closer to state-of-the-practice, we specifically invited experts who are also authors (i.e., creators) of verification detection tools to participate in the questionnaire. It is worth noting that, in line with the criteria outlined in Section 3, we exclusively considered authors of the papers that met our specified selection criteria.

Concerning the categories, we asked the experts about *Representativeness*, *Clarity*, *Comprehensiveness*, and *Usefulness* in order to assess the general quality of the whole category. Concerning the vulnerabilities, we inquired with experts to evaluate the *Clarity* and *Comprehensibility* of the titles and descriptions provided for each vulnerability. We provided questionnaire participants with all the information regarding each vulnerability and its respective categories, code samples for each vulnerability, and access to the OpenSCV website at https://openscv.dei.uc.pt.

Each question posed to participants required to be answered using a term from the following Likert scale, each associated with a numeric value: "Strongly Disagree" (-2), "Disagree" (-1), "Neutral" (0), "Agree" (1), and "Strongly Agree" (2). Subsequently, we aggregated responses from the participants and calculated the average value for each specific case. These average values were then normalized to [0,1]. A vulnerability or category with a score of greater than 0.5 indicates expert acceptance.

Due to the huge number of nodes within our taxonomy (i.e., 124 vulnerabilities and categories), we opted to break down the taxonomy into six distinct parts: Part 1 including *Unsafe External Calls*, Part 2 including *Mishandled Events, Gas Depletion, Erroneous Credit Transfer*, Part 3 including one-third of *Bad Programming and Language Practice*, Part 4 including another one-third of *Bad Programming and Language Practice*, Part5 including the final one-third of *Bad Programming and Language Practice*, and finally Part6 including *Arithmetic Issues, Improper Access Controls*. We randomly allocated these parts among participants.

We received 28 responses to a total of 150 inquiries(i.e., the response rate = 18%) from 8 different countries. We marked the demographic information as optional (i.e., attempting to increase the response rate), and because of this, we received 2 answers from non-identified countries. Participants are people involved with secure and/or reliable blockchain systems (e.g., Slither developers) occupying different positions (e.g., Director of Engineering, Postdoctoral Students, PhD students, Full Professors).

Overall, the questionnaire participants appreciated OpenSCV's categorization and hierarchical structure and mentioned that the provided information (i.e., title, description, and code excerpt) made it easier to understand the taxonomy. In terms of general aspects to improve, these mostly consisted of minor adjustments to names or descriptions, for clarity.

Regarding the questionnaire results, we received positive feedback for all categories, which means the score achieved was greater than 0.5. To be more specific, the score for *Representativeness* was 0.79, for *Clarity* was 0.78, for *Comprehensiveness* was 0.76, and finally for *Usefulness* was 0.79. Thus, the overall score for all categories was 0.78. Regarding the vulnerabilities, we also received positive feedback for both title and description. The overall score for the title was 0.77, and the overall score for the description was 0.75. Although we have received positive feedback regarding the vulnerabilities and categories of OpenSCV, we have taken the aforementioned suggestions (i.e., textual suggestions) into consideration and made minor adjustments to improve the titles and descriptions of a few vulnerabilities. Further details regarding the questionnaire results can be viewed at Vidal et al. (2024a).

5.3 Main Contributors to the Overall Quality of the Taxonomy

We now summarize the main aspects that we believe are the main contributors to the general quality of the taxonomy, which we are making publicly available at https://openscv.dei.uc.pt (Vidal et al. 2024c). In terms of organization, we opted for a *hierarchical structure*, as it may be useful from a defect prevention perspective. From a language designer's perspective, understanding that there is a certain group of defects that are related to, for instance, gas depletion may be helpful for designing effective protection mechanisms against those defects. Such mechanisms may share common strategies. From a developers' perspective, it helps identify potential issues within their smart contract code by providing tools for conducting a "top->down" analysis. For instance, it organizes information hierarchically, making it simpler to pinpoint similar problems. For instance, developers can verify the correct implementation of credit transfer functions by examining the *Erroneous Credit Transfer* category and its subcategories.

A taxonomic structure of this kind allows setting *homogeneous levels of abstraction* in an easier manner, which we iteratively tried to achieve, although this kind of goal is quite difficult as it should be balanced with the number of items and overall tree complexity (and in some cases, due to the specificity of the problem, this may not even be possible). We tried to, as much as possible, *reuse existing terminology* although many times we converged to the use of new terms (adapted from the literature), for clarity purposes. The required nomenclature adaptations integrated into our taxonomy were carried out mostly with the goal of making the items *non-ambiguous* (and *uniquely identifiable* also) and also fostering the *determinism of the classification* process by clarifying the meaning of each vulnerability. We complemented this with the available information from DASP, SWC, Ramederet al. (2022), and CWE, targeting to make the taxonomy further *comprehensible* and *non-ambiguous* (multiple perspectives will dissipate standing doubts, fostering *repeatability*).

The taxonomy construction process involved the analysis of a relatively large number of papers, tools, and other classifications, with the main goal of fostering *completeness* (i.e., good coverage), which in the end makes it also more *useful* as we end up forming a unified view of the landscape of smart contract vulnerabilities. As previously mentioned, we found that the number of papers and respective vulnerabilities analyzed (i.e., an initial set of 481 vulnerabilities collected from 77 papers) was actually a main contributor to the overall quality of the taxonomy, with a few late additions becoming trivial to map. It is worthwhile

mentioning that the created structure is flexible in the sense that we make it *open to the community* and, in particular, open to community contributions, which can be carried out by submitting issue requests at the OpenSCV GitHub repository (Vidal et al. 2024b).

OpenSCV associates each type of vulnerability with a corresponding CWE entry, offering a comprehensive overview of each vulnerability. It also includes Defect Type/Qualifier mapping from ODC, which provides insight into the nature of each vulnerability. For an in-depth analysis, OpenSCV links each issue to prominent community classifications such as SWC and DASP and offers vulnerable code snippets along with their respective fixes. Additionally, it features complete Solidity programs within its dataset.

In addition to the above aspects of OpenSCV, we have also included the latest compiler version that can automatically detect or eliminate a specific vulnerability. This helps developers to mitigate vulnerabilities by upgrading to the updated compiler version. Moreover, we provide a mapping between each vulnerability and vulnerability detection tools. This will also help developers in identifying the most effective combination of tools to detect vulnerabilities within their code. Furthermore, developers can benefit from the OpenSCV because it fills a gap in name standardization, allowing easier comparisons of results from different vulnerability detection tools. For future work, we are planning to provide an API, which would allow the tools to be integrated with our taxonomy, offering a unifying service for naming.

6 Threats to Validity

This section discusses the main threats to the validity of this work. To minimize the chances of creating an *incorrect structure or providing incorrect vulnerability information*, we formalized the taxonomy creation process, which was based on several quality criteria identified in the state of the art, and especially made use of several researchers (i.e., one Early Stage Researcher and 2 Experienced Researchers) who incrementally and iteratively built the taxonomy following a bottom-up approach. The process was enriched by establishing relations to other classifications in the blockchain context (i.e., SWC, DASP, Rameder et al. (2022)) and in a more general context (i.e., CWE). We also characterized each vulnerability using ODC and an example, which also served to minimize doubts and clear divergences among researchers. In addition, we provide the taxonomy as a live structure at Vidal et al. (2024c) supported by a GitHub repository (Vidal et al. 2024b) so that possible mistakes are corrected and also allow future updates, changes, and overall taxonomy evolution.

We are aware that *a classification or categorization scheme or a taxonomy may assume one of several possible forms*: we may have more or fewer categories, we may have a deeper tree, the organization may or may not be hierarchical, and so on. While such diversity is acceptable (as long as the organization and individual items are correct), we opted to focus on the taxonomy creation process instead of on forcing a certain structure. For this purpose, we identified quality criteria and analyzed similar structures in the state of the art so that we could learn from possible mistakes and incorporate lessons learned by previous researchers. While the current structure is a proposal, we made it open to change and evolve by opening it to the community and also by directly providing 'Request For Change' templates to facilitate changes or additions to the present form.

An important aspect is that the taxonomy creation process was guided by the research that was found during the analysis of the state of the art. Thus, we may have missed some relevant work in this context, and with time, this gap may become greater. For instance,

the fact that we focus on vulnerability detection research may lead us to miss other types of studies (e.g., empirical studies) which may also introduce new vulnerabilities, despite not proposing a specific vulnerability detection technique or tool. The fact that we were already aware of contributions coming from 3 areas: research on vulnerability classification, initiatives on vulnerability classification that are community-oriented, and research on vulnerability detection, allowed for a more efficient search, through which we believe captured representative research in this context. Despite this, and to mitigate possible gaps between the set of works considered to build OpenSCV and the set not captured during the collection of papers in this work, we prepared a supporting infrastructure to allow continuous updates and evolution of OpenSCV. Thus, we are now able to easily identify and integrate new research in vulnerability detection that may bring in emerging smart contract vulnerabilities.

7 Conclusion

In this paper, we presented an open hierarchical taxonomy for smart contract vulnerabilities. The taxonomy is up-to-date according to the current state of the practice and is prepared to handle future modifications and evolution. To build the taxonomy, we began by analyzing current vulnerability classification schemes for blockchain. We also analyzed the announced detection capabilities of the works on smart contract vulnerability detection. We then followed an iterative process to construct the OpenSCV taxonomy. We discussed the proposed taxonomy characteristics and coverage against the state of the practice. In particular, we analyzed the announced detection ability of current industry-level tools and mapped it to the OpenSCV taxonomy. We then validated our taxonomy by carrying out a questionnaire involving experts on blockchain and smart contract security. In future work, we plan on using this taxonomy as a basis to define a benchmark for smart contract vulnerability detection tools.

Acknowledgements This work is funded by the FCT - Foundation for Science and Technology, I.P./MCTES through national funds (PIDDAC), within the scope of CISUC R&D Unit - UIDB/00326/2020 or project code UIDP/00326/2020; by Project "Agenda Mobilizadora Sines Nexus". ref. No. 7113, supported by the Recovery and Resilience Plan (PRR) and by the European Funds Next Generation EU, following Notice No. 02/C05-i01/2022, Component 5 - Capitalization and Business Innovation - Mobilizing Agendas for Business Innovation; and by Foundation for Science and Technology (FCT), Grant Nr. 2023.03131.BD.

Funding Open access funding provided by FCT|FCCN (b-on).

Data Availability All data used to write this paper is available at Zenodo (Vidal et al. 2024a) and linked to Github (Vidal et al. 2024b). A visual representation of the taxonomy is available at https://openscv.dei.uc.pt (Vidal et al. 2024c).

Declarations

Conflicts of interest The authors declare that they have no conflict of interest.

References

Agbo C, Mahmoud Q, Eklund J (2019) Blockchain technology in healthcare: a systematic review. Healthcare 7(2):56. https://doi.org/10.3390/healthcare7020056. https://www.mdpi.com/2227-9032/

Akca S, Rajan A, Peng C (2019) SolAnalyser: a framework for analysing and testing smart contracts. In: 2019 26th Asia-Pacific software engineering conference (APSEC), IEEE, Putrajaya, Malaysia, pp 482–489. https://doi.org/10.1109/APSEC48747.2019.00071. https://ieeexplore.ieee.org/document/8945725/

Amiet N (2021) Blockchain vulnerabilities in practice. Digital Threats: Research and Practice 2(2):1–7. https://doi.org/10.1145/3407230

Amoroso EG (1994) Fundamentals of computer security technology. Prentice-Hall Inc, USA

Antonopoulos A, Wood G (2018) Mastering Ethereum: Building Smart Contracts and DApps. O'Reilly Media, Inc

Argañaraz MC, Berón MM, Pereira MJV, Henriques PR (2020) Detection of vulnerabilities in smart contracts specifications in ethereum platforms. In: 9th Symposium on languages, applications and technologies (SLATE 2020), Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Barcelos, Portugal, OpenAccess Series in Informatics (OASIcs), p 16. https://doi.org/10.4230/OASIcs.SLATE.2020.0

Ashizawa N, Yanai N, Cruz JP, Okamura S (2021) Eth2Vec: learning contract-wide code representations for vulnerability detection on ethereum smart contracts. In: Proceedings of the 3rd ACM international symposium on blockchain and secure critical infrastructure, ACM, New York, USA, pp 47–59. https://doi.org/10.1145/3457337.3457841

Ashouri M (2020) Etherolic. In: Proceedings of the 35th annual ACM symposium on applied computing, ACM, New York, USA, pp 353–356. https://doi.org/10.1145/3341105.3374226

Ashraf I, Ma X, Jiang B, Chan WK (2020) GasFuzzer: fuzzing ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities. IEEE Access 8:99552–99564. https://doi.org/10.1109/ACCESS.2020.2995183

Atzei N, Bartoletti M, Cimoli T (2017) A survey of attacks on ethereum smart contracts (SoK) 164–186. https://doi.org/10.1007/978-3-662-54455-6_8

Avizienis A, Laprie JC, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secure Comput 1(1):11–33. https://doi.org/10.1109/TDSC.2004.2. http://ieeexplore.ieee.org/document/1335465/

Ayoade G, Bauman E, Khan L, Hamlen K (2019) Smart contract defense through bytecode rewriting. In: 2019 IEEE International conference on blockchain (Blockchain), IEEE, Atlanta, GA, USA, pp 384–389. https://doi.org/10.1109/Blockchain.2019.00059. https://ieeexplore.ieee.org/document/8946210/

Bishop M, Bailey D (1996) A Critical Analysis of Vulnerability Taxonomies. Tech. rep. https://apps.dtic.mil/sti/citations/ADA453251

Blockstack A (2021) Clarity. https://github.com/clarity-lang

Bose P, Das D, Chen Y, Feng Y, Kruegel C, Vigna G (2022) SAILFISH: vetting smart contract state-inconsistency bugs in seconds. In: 2022 IEEE symposium on security and privacy (SP), IEEE, San Francisco, CA, USA, pp 161–178. https://doi.org/10.1109/SP46214.2022.9833721. https://ieeexplore.ieee.org/document/9833721/

Brent L, Grech N, Lagouvardos S, Scholz B, Smaragdakis Y (2020) Ethainter: a smart contract security analyzer for composite vulnerabilities. In: Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation, Association for Computing Machinery, New York, USA, PLDI 2020, pp 454–469. https://doi.org/10.1145/3385412.3385990

Brent L, Jurisevic A, Kong M, Liu E, Gauthier F, Gramoli V, Holz R, Scholzm B (2018) Vandal: a scalable security analysis framework for smart contracts. https://arxiv.org/pdf/1809.03981v1.pdf

Chang J, Gao B, Xiao H, Sun J, Cai Y, Yang Z (2019) sCompile: critical path identification and analysis for smart contracts. In: Ait-Ameur Y, Qin S (eds) Formal Methods and Software Engineering. Springer International Publishing, Cham, pp 286–304

Chen T, Cao R, Li T, Luo X, Gu G, Zhang Y, Liao Z, Zhu H, Chen G, He Z, Tang Y, Lin X, Zhang X (2020b) SODA: a generic online detection framework for smart contracts. In: Proceedings 2020 network and distributed system security symposium, internet society, Reston, VA. https://doi.org/10.14722/ndss.2020.24449. https://www.ndss-symposium.org/wp-content/uploads/2020/02/24449.pdf

Chen T, Feng Y, Li Z, Zhou H, Luo X, Li X, Xiao X, Chen J, Zhang X (2021) GasChecker: scalable analysis for discovering gas-inefficient smart contracts. IEEE Trans Emerg Topics Comput 9(3):1433–1448. https://doi.org/10.1109/TETC.2020.2979019

Chen W, Sun Z, Wang H, Luo X, Cai H, Wu L (2022) WASAI: uncovering vulnerabilities in Wasm smart contracts. In: Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis, ACM, New York, USA, pp 703–715. https://doi.org/10.1145/3533767.3534218

Chen J, Xia X, Lo D, Grundy J, Luo DX, Chen T (2020) Defining smart contract defects on ethereum. IEEE Trans Softw Eng 1. https://doi.org/10.1109/TSE.2020.2989002

Chinen Y, Yanai N, Cruz JP, Okamura S (2020) RA: hunting for re-entrancy attacks in ethereum smart contracts via static analysis. In: 2020 IEEE International conference on blockchain (Blockchain), IEEE, Rhodes, Greece, pp 327–336. https://doi.org/10.1109/Blockchain50366.2020.00048. https://ieeexplore.ieee.org/document/9284679/

Choi J, Kim D, Kim S, Grieco G, Groce A, Cha SK (2021) SMARTIAN: enhancing smart contract fuzzing with static and dynamic data-flow analyses. In: 2021 36th IEEE/ACM international conference on automated software engineering (ASE), IEEE, pp 227–239. https://doi.org/10.1109/ASE51524.2021.9678888

Clarivate (2021) Journal Citation Reports (JCR). http://jcr.clarivate.com

Coblenz M (2019) The Obsidian Smart Contract Language. https://obsidian.readthedocs.io/en/latest/

ConsenSys (2021) Mythril. https://github.com/ConsenSys/mythril

Crincoli G, Iadarola G, La Rocca PE, Martinelli F, Mercaldo F, Santone A (2022) Vulnerable smart contract detection by means of model checking. In: Proceedings of the Fourth ACM international symposium on blockchain and secure critical infrastructure, ACM, New York, USA, pp 3–10. https://doi.org/10.1145/3494106.3528672

Cui S, Zhao G, Gao Y, Tavu T, Huang J (2022) VRust. In: Proceedings of the 2022 ACM SIGSAC conference on computer and communications security, ACM, New York, USA, pp 639–652. https://doi.org/10.1145/3548606.3560552

CWE Community (2009) Common Weakness Enumeration. https://cwe.mitre.org/about/index.html

di Angelo M, Salzer G (2019) A Survey of tools for analyzing ethereum smart contracts. In: 2019 IEEE International conference on decentralized applications and infrastructures (DAPPCON), IEEE, Newark, CA, USA, pp 69–78. https://doi.org/10.1109/DAPPCON.2019.00018. https://ieeexplore.ieee.org/document/8782988/

Ding M, Li P, Li S, Zhang H (2021) HFContractFuzzer: fuzzing hyperledger fabric smart contracts for vulnerability detection. In: Evaluation and assessment in software engineering, ACM, New York, USA, pp 321–328. https://doi.org/10.1145/3463274.3463351

Durieux T, Ferreira JF, Abreu R, Cruz P (2020) Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, Association for Computing Machinery, New York, USA, ICSE '20, pp 530–541. https://doi.org/10.1145/3377811.3380364

Eshghie M, Artho C, Gurov D (2021) Dynamic vulnerability detection on smart contracts using machine learning. In: Evaluation and assessment in software engineering, ACM, New York, USA, pp 305–312. https://doi.org/10.1145/3463274.3463348

Ethereum's Github (2022) Pure Issue. https://github.com/ethereum/solidity/issues/13174

Feist J, Grieco G, Groce A (2019) Slither: a static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd International workshop on emerging trends in software engineering for blockchain (WETSEB), IEEE, Montreal, QC, Canada, WETSEB '19, pp 8–15. https://doi.org/10.1109/WETSEB.2019.00008. https://ieeexplore.ieee.org/document/8823898/

Fu M, Wu L, Hong Z, Zhu F, Sun H, Feng W (2019) A critical-path-coverage-based vulnerability detection method for smart contracts. IEEE Access 7:147327–147344. https://doi.org/10.1109/ACCESS.2019.2947146

Gao J, Liu H, Liu C, Li Q, Guan Z, Chen Z (2019) EASYFLOW: keep ethereum away from overflow. In: 2019 IEEE/ACM 41st International conference on software engineering: companion proceedings (ICSE-Companion), IEEE, Montreal, QC, Canada, pp 23–26. https://doi.org/10.1109/ICSE-Companion.2019.00029. https://ieeexplore.ieee.org/document/8802775/

Geneiatakis D, Soupionis Y, Steri G, Kounelis I, Neisse R, Nai-Fovino I (2020) Blockchain performance analysis for supporting cross-border e-government services. IEEE Trans Eng Manag 67(4):1310–1322. https://doi.org/10.1109/TEM.2020.2979325. https://ieeexplore.ieee.org/document/9102377/

Ghaleb A, Pattabiraman K (2020) 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, association for computing machinery, New York, USA, ISSTA 2020, pp 415–427. https://doi.org/10.1145/3395363.3397385

Ghaleb A, Rubin J, Pattabiraman K (2023) AChecker: statically detecting smart contract access control vulnerabilities. In: Proc ACM ICSE

government U (1999) National Vulnerability Database. https://nvd.nist.gov/

Grech A, Camilleri AF (2017) Blockchain in Education. Publications Office of the European Union. https://doi.org/10.2760/60649. https://ec.europa.eu/jrc/en/open-education/legal-notice

Grech N, Kong M, Jurisevic A, Brent L, Scholz B, Smaragdakis Y (2020) MadMax: analyzing the out-of-gas world of smart contracts. Commun ACM 63(10):87–95. https://doi.org/10.1145/3416262

Grishchenko I, Maffei M, Schneidewind C (2018) A semantic framework for the security analysis of ethereum smart contracts. In: Bauer L, Küsters R (eds) principles of security and trust, vol 10804, Springer International Publishing, Uppsala, Sweden, pp 243–269. https://doi.org/10.1007/978-3-319-89722-6_10

Gupta R, Patel MM, Shukla A, Tanwar S (2022) Deep learning-based malicious smart contract detection scheme for internet of things environment. Comput Electr Eng 97:107583. https://doi.org/10.1016/j.compeleceng.2021.107583

Hajdu Á, Jovanović D (2020) solc-verify: a modular verifier for solidity smart contracts. pp 161–179. https://doi.org/10.1007/978-3-030-41600-3_11

Hansman S, Hunt R (2005) A taxonomy of network and computer attacks. Computers & Security 24(1):31–43. https://doi.org/10.1016/j.cose.2004.06.011. https://www.sciencedirect.com/science/article/pii/S0167404804001804

Hartel P, Schumi R (2020) Mutation testing of smart contracts at scale. In: Ahrendt W, Wehrheim H (eds) Tests and Proofs - 14th International Conference, TAP 2020, held as part of STAF 2020, Proceedings, Lecture Notes in Computer Science(including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Springer Open, Bergen, Norway, pp 23–42. https://doi.org/10.1007/978-3-030-50995-8_2. http://link.springer.com/10.1007/978-3-030-50995-8_2

Hewa T, Ylianttila M, Liyanage M (2021) Survey on blockchain based smart contracts: applications, opportunities and challenges. J Netw Comput Appl 177:102857

He N, Zhang R, Wang H, Wu L, Luo X, Guo Y, Yu T, Jiang X (2021) {EOSAFE}: security analysis of {EOSIO} smart contracts. In: 30th USENIX security symposium (USENIX Security 21), pp 1271–1288

Howard JD (1997) An analysis of security incidents on the Internet 1989-1995. PhD thesis, Carnegie Mellon University, USA. https://www.proquest.com/openview/26b4425b41777ee9b6cac10b78da998a/1?pq-origsite=gscholar&cbl=18750&diss=y

Hu B, Zhang Z, Liu J, Liu Y, Yin J, Lu R, Lin X (2021) A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems. Patterns 2(2):100179. https://doi.org/10.1016/j.patter.2020.100179

Hu T, Li B, Pan Z, Qian C (2023) Detect defects of solidity smart contract based on the knowledge graph. IEEE Trans Reliab 1–17. https://doi.org/10.1109/TR.2023.3233999. https://ieeexplore.ieee.org/document/10025570/

Hwang SJ, Choi SH, Shin J, Choi YH (2022) CodeNet: code-targeted convolutional neural network architecture for smart contract vulnerability detection. IEEE Access 10:32595–32607. https://doi.org/10.1109/ACCESS.2022.3162065

I Group et al (2010) IEEE Standard Classification for Software Anomalies. IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993, pp 1–23. https://doi.org/10.1109/IEEESTD.2010.5399061

IBM (2013a) Orthogonal Defect Classification v 5.2 Extensions for GUI, User Documentation, Build & NLS. https://s3.us.cloud-object-storage.appdomain.cloud/res-files/70-ODC-5-2-Extensions.pdf

IBM (2013b) Orthogonal Defect Classification v 5.2 for Software Design and Code. https://s3.us.cloud-object-storage.appdomain.cloud/res-files/70-ODC-5-2.pdf

Ivanov N, Li C, Yan Q, Sun Z, Cao Z, Luo X (2023) Security Threat Mitigation For Smart Contracts: A Comprehensive Survey. ACM Comput Surv. https://doi.org/10.1145/3593293

Jiang B, Liu Y, Chan WK (2018) ContractFuzzer: fuzzing smart contracts for vulnerability detection. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, Association for Computing Machinery, New York, USA, ASE 2018, pp 259–269. https://doi.org/10.1145/3238147.3238177

Ji R, He N, Wu L, Wang H, Bai G, Guo Y (2020) DEPOSafe: demystifying the fake deposit vulnerability in Ethereum smart contracts. In: 2020 25th international conference on engineering of complex computer systems (ICECCS), IEEE, pp 125–134. https://doi.org/10.1109/ICECCS51672.2020.00022. https://ieeexplore.ieee.org/document/9376204/

Jin L, Cao Y, Chen Y, Zhang D, Campanoni S (2023) ExGen: cross-platform, automated exploit generation for smart contract vulnerabilities. IEEE Trans Dependable Secure Comput 20(1):650–664. https://doi.org/10.1109/TDSC.2022.3141396

Kaleem M, Mavridou A, Laszka A (2020) Vyper: a security comparison with solidity based on common Vulnerabilities. In: 2020 2nd conference on blockchain research & applications for innovative networks and services (BRAINS), IEEE, pp 107–111. https://doi.org/10.1109/BRAINS49436.2020.9223278

Kalra S, Goel S, Dhawan M, Sharma S (2018) ZEUS: analyzing safety of smart contracts. In: Proceedings 2018 network and distributed system security symposium, Internet Society, Reston, VA, pp 2018–02. https://doi.org/10.14722/ndss.2018.23082. https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_09-1_Kalra_paper.pdf

Khan S, Amin MB, Azar AT, Aslam S (2021) Towards interoperable blockchains: a survey on the role of smart contracts in blockchain interoperability. IEEE Access 9:116672–116691. https://doi.org/10.1109/ACCESS.2021.3106384

Kolluri A, Nikolic I, Sergey I, Hobor A, Saxena P (2019) Exploiting the laws of order in smart contracts. In: Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, Association for Computing Machinery, New York, USA, ISSTA 2019, pp 363–373. https://doi.org/10.1145/3293882.3330560

Krsul IV (1998) Software vulnerability analysis. PhD thesis, Purdue University. https://www.proquest.com/openview/10fa0675998eeecf99bbc64ca3a46650/1?pq-origsite=gscholar&cbl=18750&diss=y

Krupp J, Rossow C (2018) TEETHER: gnawing at ethereum to automatically exploit smart contracts. In: Proceedings of the 27th USENIX Conference on Security Symposium, USENIX Association, USA, SEC'18, pp 1317–1333

Liao JW, Tsai TT, He CK, Tien CW (2019) SoliAudit: smart contract vulnerability assessment based on machine learning and fuzz testing. In: 2019 Sixth international conference on internet of things: systems, management and security (IOTSMS), IEEE, Granada, Spain, pp 458–465. https://doi.org/10.1109/IOTSMS48152.2019.8939256. https://ieeexplore.ieee.org/document/8939256/

Liao Z, Zheng Z, Chen X, Nan Y (2022) SmartDagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability. In: Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis, ACM, New York, USA, pp 752–764. https://doi.org/10.1145/3533767.3534222

Li W, He J, Zhao G, Yang J, Li S, Lai R, Li P, Tang H, Luo H, Zhou Z (2022c) EOSIOAnalyzer: an effective static analysis vulnerability detection framework for EOSIO smart contracts. In: 2022 IEEE 46th annual computers, software, and applications conference (COMPSAC), IEEE, Los Alamitos, CA, USA, pp 746–756. https://doi.org/10.1109/COMPSAC54236.2022.00124. https://ieeexplore.ieee.org/document/9842620/

Li P, Li S, Ding M, Yu J, Zhang H, Zhou X, Li J (2022b) A vulnerability detection framework for hyperledger fabric smart contracts based on dynamic and static analysis. In: The International Conference on Evaluation and Assessment in Software Engineering 2022, ACM, New York, USA, pp 366–374. https://doi.org/10.1145/3530019.3531342

Li Z, Lu S, Zhang R, Xue R, Ma W, Liang R, Zhao Z, Gao S (2022) SmartFast: an accurate and robust formal analysis tool for Ethereum smart contracts. Empir Softw Eng 27(7):197. https://doi.org/10.1007/s10664-022-10218-2

Lindqvist U, Jonsson E (1997) How to systematically classify computer security intrusions. pp 154–163

Li B, Pan Z, Hu T (2022) ReDefender: detecting Reentrancy Vulnerabilities in Smart Contracts Automatically. IEEE Trans Reliab 71(2):984–999. https://doi.org/10.1109/TR.2022.3161634

Liu C, Liu H, Cao Z, Chen Z, Chen B, Roscoe B (2018) ReGuard: finding reentrancy Bugs in smart Contracts. In: Proceedings of the 40th international conference on software engineering: companion proceeedings, ACM, New York, USA, pp 65–68. https://doi.org/10.1145/3183440.3183495

Liu Z, Qian P, Wang X, Zhuang Y, Qiu L, Wang X (2021) Combining graph neural networks with expert knowledge for smart contract vulnerability detection. IEEE Trans Knowl Data Eng 35(2):1–1. https://doi.org/10.1109/TKDE.2021.3095196. https://ieeexplore.ieee.org/document/9477066/

Lough DL (2001) A taxonomy of computer attacks with applications to wireless networks. PhD thesis, Virginia Polytechnic Institute and State University

Luu L, Chu DH, Olickel H, Saxena P, Hobor A (2016) Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, Association for Computing Machinery, New York, USA, CCS '16, pp 254–269. https://doi.org/10.1145/2976749.2978309

Lu N, Wang B, Zhang Y, Shi W, Esposito C (2019) NeuCheck: a more practical Ethereum smart contract security analysis tool. Software: Practice and Experience n/a(n/a):1–20. https://doi.org/10.1002/spe.2745

Mann DE, Christey SM (1999) Towards a common enumeration of vulnerabilities. In: 2nd Workshop on research with security vulnerability databases, Purdue University in West Lafayette, Indiana, pp 1–13

Manning A (2018) Solidity security: comprehensive list of known attack vectors and common anti-patterns. https://github.com/sigp/solidity-security-blog

Ma F, Ren M, Ouyang L, Chen Y, Zhu J, Chen T, Zheng Y, Dai X, Jiang Y, Sun J (2023) Pied-Piper: revealing the backdoor threats in ethereum ERC token contracts. ACM Trans Softw Eng Methodol 32(3):1–24. https://doi.org/10.1145/3560264

Mavridou A, Laszka A, Stachtiari E, Dubey A (2019) VeriSolid: correct-by-design smart contracts for Ethereum. In: Goldberg I, Moore T (eds) financial cryptography and data security. Springer International Publishing, Cham, pp 446–465

Mavridou A, Laszka A (2018) Designing secure Ethereum smart contracts: a finite State machine based approach. In: Meiklejohn S, Sako K (eds) financial cryptography and data security, Springer Berlin Heidel-

berg, pp 523–540. https://www.springerprofessional.de/en/designing-secure-ethereum-smart-contracts-a-finite-state-machine/17118720

Ma F, Xu Z, Ren M, Yin Z, Chen Y, Qiao L, Gu B, Li H, Jiang Y, Sun J (2022) Pluto: exposing vulnerabilities in inter-contract scenarios. IEEE Trans Softw Eng 48(11):4380–4396. https://doi.org/10.1109/TSE.2021.3117966. https://ieeexplore.ieee.org/document/9562567/

MITRE Corporation (1999) Common Vulnerabilities and Exposures. https://www.cve.org/

Mi F, Wang Z, Zhao C, Guo J, Ahmed F, Khan L (2021) VSCL: automating vulnerability detection in smart contracts with deep learning. In: 2021 IEEE international conference on blockchain and cryptocurrency (ICBC), IEEE, Sydney, Australia, pp 1–9. https://doi.org/10.1109/ICBC51069.2021.9461050. https://ieeexplore.ieee.org/document/9461050/

Momeni P, Wang Y, Samavi R (2019) Machine learning model for smart contracts security analysis. In: 2019 17th international conference on privacy, security and trust (PST), IEEE, Fredericton, NB, Canada, pp 1–6. https://doi.org/10.1109/PST47121.2019.8949045. https://ieeexplore.ieee.org/document/8949045/

Nassirzadeh B, Sun H, Banescu S, Ganesh V (2023) Gas Gauge: a security analysis tool for smart contract out-of-gas vulnerabilities. In: Mathematical Research for Blockchain Economy. Springer International Publishing, Cham, pp 143–167

NCC Group (2019) DASP. https://dasp.co/

NCCGroup (2021) Decentralized Application Security Project (DASP) Top10. https://dasp.co/

Nguyen TD, Pham LH, Sun J, Lin Y, Minh QT (2020) SFuzz: an efficient adaptive fuzzer for solidity smart contracts. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, Association for Computing Machinery, New York, USA, ICSE '20, pp 778–788. https://doi.org/10.1145/3377811.3380334

Nishida Y, Saito H, Ran C, Akira K, Jun F, Kohei S, Atsushi I (2021) Helmholtz: A verifier for Tezos smart contracts cased on refinement types. In: Groote JF, Larsen KG (eds) tools and algorithms for the construction and analysis of systems. Springer International Publishing, Cham, pp 262–280

OWASP Foundation (2001) OWASP. https://owasp.org/www-community/vulnerabilities/#

Pani S, Nallagonda HV, Vigneswaran, Medicherla RK, Rajan M (2023) SmartFuzzDriverGen: smart contract fuzzing automation for Golang. In: 16th innovations in software engineering conference, ACM, New York, USA, pp 1–11. https://doi.org/10.1145/3578527.3578538

Qian P, Liu ZG, He QM, Huang BT, Tian DZ, Wang X (2022) Smart contract vulnerability detection technique: a survey. Ruan Jian Xue Bao/Journal of Software 33(8):3059–3085. https://doi.org/10.13328/j.cnki.jos.006375. arXiv:2209.05872

Rameder H, di Angelo M, Salzer G (2022) Review of automated vulnerability analysis of smart contracts on Ethereum. Frontiers in Blockchain 5. https://doi.org/10.3389/fbloc.2022.814977

Rodler M, Li W, Karame GO, Davi L (2019) Sereum: protecting existing smart contracts against re-entrancy attacks. In: Proceedings 2019 network and distributed system security symposium, internet society, Reston, VA. https://doi.org/10.14722/ndss.2019.23413. https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_09-3_Rodler_paper.pdf

Shakya S, Mukherjee A, Halder R, Maiti A, Chaturvedi A (2022) SmartMixModel: machine learning-based vulnerability detection of solidity smart contracts. In: 2022 IEEE international conference on blockchain (blockchain), IEEE, Espoo, Finland, pp 37–44. https://doi.org/10.1109/Blockchain55522.2022.00016. https://ieeexplore.ieee.org/document/9881798/

Siegel D (2016) Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists

Slither's Github (2019) Slither Vulnerabilities Detection. https://github.com/crytic/slither

SmartContractSecurity (2020) Smart Contract Weakness Classification (SWC) and Test Cases. http://swcregistry.io/

SmartDec Corporation (2018) SmartDec - Classification of smart contract vulnerabilities. https://github.com/smartdec/classification

So S, Lee M, Park J, Lee H, Oh H (2020) VERISMART: a highly precise safety verifier for Ethereum smart contracts. In: 2020 IEEE symposium on Security and Privacy (SP), IEEE, San Francisco, CA, USA, pp 1678–1694. https://doi.org/10.1109/SP40000.2020.00032. https://ieeexplore.ieee.org/document/9152689/

Solidity (2023) Solidity Documentation 0.8.17. https://docs.soliditylang.org/en/v0.8.17/contracts.html

Song J, He H, Lv Z, Su C, Xu G, Wang W (2019) An efficient vulnerability detection model for Ethereum smart contracts. In: Liu JK, Huang X (ed) network and system security, Springer International Publishing, Cham, pp 433–442. https://doi.org/10.1007/978-3-030-36938-5_26

Staderini M, Palli C, Bondavalli A (2020) Classification of Ethereum vulnerabilities and their propagations. In: 2020 second international conference on blockchain computing and applications (BCCA), IEEE, pp 44–51. https://doi.org/10.1109/BCCA50787.2020.9274458. https://ieeexplore.ieee.org/document/9274458/

Staderini M, Patariciza A, Bondavalli A (2022) Security evaluation and improvement of solidity smart contracts. SSRN Electron J. https://doi.org/10.2139/ssrn.4038087

Stephens J, Ferles K, Mariano B, Lahiri S, Dillig I (2021) SmartPulse: automated checking of temporal properties in smart contracts. In: 2021 IEEE symposium on security and privacy (SP), IEEE, San Francisco, CA, USA, pp 555–571. https://doi.org/10.1109/SP40001.2021.00085. https://ieeexplore.ieee.org/document/9519387/

Sunbeom S, Seongjoon H, Hakjoo O (2021) Smartest: effectively hunting vulnerable transaction sequences in smart contracts through language modelguided symbolic execution. In: in 30th USENIX Security Symposium, USENIX Association. https://www.usenix.org/system/files/sec21-so.pdf

Sun X, Tu L, Zhang J, Cai J, Li B, Wang Y (2023) ASSBert: active and semi-supervised bert for smart contract vulnerability detection. J Inf Secur Appl 73:103423. https://doi.org/10.1016/j.jisa.2023.103423

The Computing Research and Education Association of Australasia (2021) CORE Conference Ranking. http://portal.core.edu.au/conf-ranks/

Tikhomirov S, Voskresenskaya E, Ivanitskiy I, Takhaviev R, Marchenko E, Alexandrov Y (2018) SmartCheck: static analysis of Ethereum smart contracts. In: Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain, ACM, New York, USA, pp 9–16. https://doi.org/10.1145/3194113.3194115

Torres CF, Iannillo AK, Gervais A, State R (2021) ConFuzzius: a data dependency-aware hybrid fuzzer for smart contracts. In: 2021 IEEE European symposium on security and privacy (EuroS&P), IEEE, Vienna, Austria, pp 103–119. https://doi.org/10.1109/EuroSP51992.2021.00018. https://ieeexplore.ieee.org/document/9581164/

Torres CF, Schütte J, State R (2018) Osiris: hunting for integer bugs in Ethereum smart contracts. In: Proceedings of the 34th annual computer security applications conference, association for computing machinery, New York, USA, ACSAC '18, pp 664–676. https://doi.org/10.1145/3274694.3274737

Tsankov P (2018) Securify2. https://github.com/eth-sri/securify2

Tsankov P, Dan A, Drachsler-Cohen D, Gervais A, Bünzli F, Vechev M (2018) Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, Association for Computing Machinery, New York, USA, CCS '18, pp 67–82. https://doi.org/10.1145/3243734.3243780

Vidal F, Ivaki N, Laranjeiro N (2024a) OpenSCV: an open hierachical taxonomy for smart contract vulnerabilities - supplemental material. https://doi.org/10.5281/zenodo.7763982

Vidal F, Ivaki N, Laranjeiro N (2024b) OpenSCV Github Repository. https://github.com/blockchain-dei/openscv

Vidal F, Ivaki N, Laranjeiro N (2024c) OpenSCV Website. https://openscv.dei.uc.pt

Vogelsteller F, Buterin V (2015) ERC20 standard. https://github.com/ethereum/eips/issues/20

Wagner G (2018) EIP-1470: Smart Contract Weakness Classification (SWC), https://github.com/ethereum/EIPs/issues/1469

Wang Z, Wen B, Ziqiang L, Shaojie L (2021) M-A-R: a dynamic symbol execution detection method for smart contract reentry vulnerability. In: Dai H-N, Liu X, Xiapu LD, Jiang X, Xiangping C (eds) blockchain and trustworthy systems. Springer, Singapore, pp 418–429

Wang H, Li Y, Lin SW, Ma L, Liu Y (2019) VULTRON: catching vulnerable smart contracts once and for all. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), IEEE, Montreal, QC, Canada, pp 1–4. https://doi.org/10.1109/ICSE-NIER.2019.00009. https://ieeexplore.ieee.org/document/8805696/

Wang W, Song J, Xu G, Li Y, Wang H, Su C (2021) ContractWard: automated vulnerability detection models for Ethereum smart contracts. IEEE Trans Network Sci Eng 8(2):1133–1144. https://doi.org/10.1109/TNSE.2020.2968505. https://ieeexplore.ieee.org/document/8967006/

Wu H, Zhang Z, Wang S, Lei Y, Lin B, Qin Y, Zhang H, Mao X (2021) Peculiar: smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), IEEE, Wuhan, China, pp 378–389. https://doi.org/10.1109/ISSRE52982.2021.00047. https://ieeexplore.ieee.org/document/9700296/

Xing C, Chen Z, Chen L, Guo X, Zheng Z, Li J (2020) A new scheme of vulnerability analysis in smart contract with machine learning. Wireless Networks. https://doi.org/10.1007/s11276-020-02379-z. https://doi.org/10.1007/s11276-020-02379-z

Xi R, Pattabiraman K (2023) A large-scale empirical study of low-level function use in Ethereum smart contracts and automated replacement. Software: Practice and Experience 53(3):631–664. https://doi.org/10.1002/spe.3163

Xue Y, Ma M, Lin Y, Sui Y, Ye J, Peng T (2020) Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In: 2020 35th IEEE/ACM international conference on automated software engineering (ASE), pp 1029–1040

Xue Y, Ye J, Zhang W, Sun J, Ma L, Wang H, Zhao J (2022) xFuzz: machine learning guided cross-contract fuzzing. IEEE Transactions on Dependable and Secure Computing pp 1–14. https://doi.org/10.1109/TDSC.2022.3182373. https://ieeexplore.ieee.org/document/9795233/

Yaga D, Mell P, Roby N, Scarfone K (2018) Blockchain technology overview. Tech. rep., National Institute of Standards and Technology, Gaithersburg, MD. https://doi.org/10.6028/NIST.IR.8202. https://nvlpubs.nist.gov/nistpubs/ir/2018/NIST.IR.8202.pdf

Ye J, Ma M, Lin Y, Ma L, Xue Y, Zhao J (2022) Vulpedia: detecting vulnerable ethereum smart contracts via abstracted vulnerability signatures. J Syst Software 192:111410. https://doi.org/10.1016/j.jss.2022.111410

Yosifova VK, Bontchev VV (2021) Possible instant messaging malware attack using unicode right-to-left override. pp 179–191. https://doi.org/10.1007/978-3-030-65722-2_11

Yu X, Zhao H, Hou B, Ying Z, Wu B (2021) DeeSCVHunter: a deep learning-based framework for smart contract vulnerability detection. In: 2021 International Joint Conference on Neural Networks (IJCNN), IEEE, Shenzhen, China, pp 1–8. https://doi.org/10.1109/IJCNN52387.2021.9534324. https://ieeexplore.ieee.org/document/9534324/

Zeng Q, He J, Zhao G, Li S, Yang J, Tang H, Luo H (2022) EtherGIS: a vulnerability detection framework for Ethereum smart contracts based on graph learning features. In: 2022 IEEE 46th annual computers, software, and applications conference (COMPSAC), IEEE, Los Alamitos, CA, USA, pp 1742–1749. https://doi.org/10.1109/COMPSAC54236.2022.00277. https://ieeexplore.ieee.org/document/9842713/

Zhang Z, Lei Y, Yan M, Yu Y, Chen J, Wang S, Mao X (2022c) Reentrancy vulnerability detection and localization: a deep learning based two-phase approach. In: Proceedings of the 37th IEEE/ACM international conference on automated software engineering, ACM, New York, USA, pp 1–13. https://doi.org/10.1145/3551349.3560428

Zhang Q, Wang Y, Li J, Ma S (2020b) EthPloit: from fuzzing to efficient exploit generation against smart contracts. In: 2020 IEEE 27th International conference on software analysis, evolution and reengineering (SANER), IEEE, London, ON, Canada, pp 116–126. https://doi.org/10.1109/SANER48275.2020.9054822. https://ieeexplore.ieee.org/document/9054822/

Zhang S, Wang M, Liu Y, Zhang Y, Yu B (2022b) Multi-transaction sequence vulnerability detection for smart contracts based on inter-path data dependency. In: 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS), IEEE, Guangzhou, China, pp 616–627. https://doi.org/10.1109/QRS57517.2022.00068. https://ieeexplore.ieee.org/document/10062352/

Zhang L, Wang J, Wang W, Jin Z, Su Y, Chen H (2022a) Smart contract vulnerability detection combined with multi-objective detection. Computer Networks 217:109289. https://doi.org/10.1016/j.comnet.2022.109289

Zhang P, Xiao F, Luo X (2019) SolidityCheck : quickly detecting smart contract problems through regular expressions. arXiv:1911.09425

Zhang P, Xiao F, Luo X (2020a) A framework and dataset for bugs in Ethereum smart contracts. In: 2020 IEEE International conference on software maintenance and evolution (ICSME), IEEE, pp 139–150. https://doi.org/10.1109/ICSME46990.2020.00023

Zheng G, Gao L, Huang L, Guan J (2021) Ethereum Smart Contract Development in Solidity. Springer, Singapore. https://doi.org/10.1007/978-981-15-6218-1

Zhou H, Milani Fard A, Makanju A (2022) The State of Ethereum smart contracts security: vulnerabilities, Countermeasures, and Tool Support. J Cybersec Priv 2(2):358–378. https://doi.org/10.3390/jcp2020019

Zhou Q, Zheng K, Zhang K, Hou L, Wang X (2022b) Vulnerability Analysis of Smart Contract for Blockchain-Based IoT Applications: A Machine Learning Approach. IEEE Int Things J 9(24):24695–24707. https://doi.org/10.1109/JIOT.2022.3196269

Zhuang Y, Liu Z, Qian P, Liu Q, Wang X, He Q (2020) Smart Contract Vulnerability Detection using Graph Neural Network. In: Proceedings of the twenty-ninth international joint conference on artificial intelligence, international joint conferences on artificial intelligence organization, California, pp 3283–3290. https://doi.org/10.24963/ijcai.2020/454

Zou W, Lo D, Kochhar PS, Le XBD, Xia X, Feng Y, Chen Z, Xu B (2019) Smart Contract Development: Challenges and Opportunities. IEEE Trans Softw Eng p 1. https://doi.org/10.1109/TSE.2019.2942301