An Empirical Evaluation of the Effectiveness of Smart Contract Verification Tools

Bruno Dias, Naghmeh Ivaki, Nuno Laranjeiro
University of Coimbra
Centre for Informatics and Systems of the University of Coimbra
Department of Informatics Engineering, Portugal
brdias@student.dei.uc.pt, naghmeh@dei.uc.pt, cnl@dei.uc.pt,

2021 IEEE 26th Pacific Rim International Symposium on Dependable Computing (PRDC) | 978-1-6654-2476-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/PRDC53464.2021.00013

***Abstract*—Blockchain has become popular due to its use in cryptocurrencies and potential to support different business-critical services (e.g., financial services, retail). The smart contract is at the center of blockchain systems and is a coded specification of an agreement between interacting partners in a transaction. Like other software artifacts, smart contracts are prone to carry residual faults. As many contracts are being used to handle financial transactions, huge losses may occur if a vulnerability is exploited. Also, a faulty contract cannot be corrected once it has been deployed on the blockchain, it can only be terminated and a new one must be deployed, which aggravates the cost of deploying contracts with faults and marks the reputation of the provider. Smart contract verification tools have been emerging, but limited knowledge is available regarding their real effectiveness. In this paper, we define a smart contract defect classification scheme based on the Orthogonal Defect Classification and apply it to a contract dataset, which has been extracted from multiple sources and holds different types of defects. We use the dataset to evaluate three state of the art verification tools regarding their fault detection performance. Results show the relatively low effectiveness of the tools and their complementarity.**

***Index Terms*—Blockchain, Smart Contract, Software Vulnerability, Fault Injection, Vulnerability Detection Tools.**

I. Introduction

Blockchain systems have become very popular mostly due to their application in cryptocurrencies, but they are also being seen as a promising way to support diverse business-critical services in a wide range of domains. At the core of a blockchain system, we find *smart contracts*, which are programs stored on the blockchain that run when predetermined conditions are met. In practice, they serve to automate an agreement between parties that wish to perform a certain transaction. Nowadays, we are seeing blockchain applications in healthcare, licensing in the music industry, or payment processing in financial services. In these contexts, the presence of a software fault in a contract may bring in disastrous consequences for the parties involved, such as the well-known bug in the DAO (Decentralized Autonomous Organization) [1].

Smart contracts can be written in mainstream languages like Java, but developers usually resort to smart-contract specific languages like Solidity [2], many times without proper expertise. The software development processes in this domain tend to be informal, making Verification and Validation (V&V) activities more difficult to integrate, at least when compared to classic critical systems, where V&V fits rather well [3]. Despite the technology being relatively recent, there are already several tools for verifying smart contracts (e.g., static analysers like Slither [4], abstract interpreters like Securify [5], or tools based on symbolic execution like Mythril [6]). Such tools are recent and known to have limitations regarding their effectiveness. Indeed, previous studies have analyzed the effectiveness of smart contract verification tools, but only focus on certain classes of tools (e.g., static analysis) [4], [5], [7], or use small sets of contracts [8], or a low number of faults [9], or do not analyze tools effectiveness with respect to the different classes of faults present in contracts [10]. Other clear difficulty is having a standard classification of faults for blockchain smart contracts, as current schemes struggle to fit the specific defects arising in smart contracts [9], [11], [12].

In this paper, we define a smart contract defect classification scheme, using the Orthogonal Defect Classification [13] as structure. We analyze several sources of contract faults and classifications, namely [2], [11], [12], [14], and define a fault classification scheme for smart contracts, in which we fit known faults. We then extract contracts from existing datasets [11], [14], [15] to create a set composed of 222 contracts that fit in three groups (following the structure in [11]): i) contracts with known software defects (including vulnerabilities) of various types; ii) contracts where those defects have been corrected; and iii) contracts designed to mislead verification tools. We then analyze the effectiveness of the latest versions of three smart contract verification tools Mythril [6], Securify2 [5], and Slither [4] by running them against the three groups of contracts. Results are analyzed according to the tools' overall detection capabilities and also their false-positive and false-negative rates. We provide a view on the different types of defects the tools are able to detect and cases where they are misled, identifying clear points where tools need to be improved.

Results reflect the relatively low detection capabilities of the different tools, which detect just a fraction of the defective contracts, but also show the complementarity of the tools, which are able to detect different faults and of different types. The alerts produced by the tools also allowed us to identify new cases of faults in the analyzed smart contracts (previously unknown), although this is carried out at the expense of manually analyzing alerts, which are also produced in different magnitudes by the tools. The main contributions of this paper are the following:

2473-3105/21/$31.00 ©2021 IEEE
DOI 10.1109/PRDC53464.2021.00013

- The definition of a defect classification scheme tailored for smart contract faults, structured around the Orthogonal Defect Classification (ODC) and in the general fault models presented in [16], which we now extended to take into account specific smart contract faults [2], [11], [12], [14].
- A reusable dataset of smart contracts, which were mostly extracted from previous work [11], [14], [15], available at [17]. The selection was made to consider contracts that hold diverse types of defects (i.e., defects representing different classes of software faults), their respective fixed versions, and contracts holding diverse defect patterns, which are however not exploitable.
- A view of the overall effectiveness of different state of the art smart contract verification tools (i.e., Securify2 [5], Mythril [6], and Slither [4]), including an analysis of the tools complementarity in detecting known and unknown faults of different types, allowing for individual tool improvement and effective tool ensemble.

This paper is organized as follows. Section II presents background and related work and Section III presents the design of our experimental study. Section IV discusses the results and Section V presents the main threats to the validity of this work. Finally, Section VI concludes this paper.

II. Background and Related Work

This section overviews background on software defect classification schemes and related work on the evaluation of smart contract verification tools. There are several works on generic **defect classification schemes**, such as the IEEE Standard 1044–2009 Classification for Software Anomalies 2010, Hewlett-Packard's Defect Origins, Types and Modes (DOTM) and the Orthogonal Defect Classification (ODC) [13]. ODC is a set of analytical methods used for software development and test process analysis to characterize software defects that consists of eight orthogonal attributes [13]. Of these attributes, there are two that serve to characterize the software defect, namely *defect type* which represents the nature of the change performed to fix a certain defect (e.g., algorithm, assignment); and *qualifier*, which complements the defect type by describing the state of the code prior to the correction (e.g., missing, incorrect, or extraneous).

There are recent efforts to define or adapt software defect classification schemes for blockchain smart contracts. The Decentralized Application Security Project (DASP) [12] identifies the top 10 types of vulnerabilities that threaten the security of smart contracts. However, the categorization presented in the work is not systematical and complete. In [18], the authors use the Bugs Framework developed by the National Institute of Standards and Technology (NIST) to classify known vulnerabilities, although it was found that most of the vulnerabilities at the time, fell outside the scope of the Bugs Framework classes. This tends to aggravate with time, as new issues are discovered and must be properly characterized by a classification scheme.

Poston [19] maps the OWASP Top 10 list of web application vulnerabilities [20] to blockchain, highlighting that 9 out of 10 apply to blockchain. Zhang et al. [11] use the IEEE Standard Classification for Software Anomalies to fit smart contract defects. The standard has now been withdrawn and the classification tends to mix causes and effects at similar semantic levels (e.g., a calculation bug is expressed at the same level as Denial of Service item, which is an effect of the exploitation of a bug). Thus, having a more comprehensive way of characterizing contract defects is still open research.

Previous work has been carried out on the **evaluation of smart contract verification tools**. Tsankov et al. [5] present an abstract interpreter (Securify) that can detect 7 vulnerabilities: Reentrancy, Locked Ether, Missing input validation, Transaction ordering-dependent amount, receiver and transfer, Mishandled exceptions and Lost Ether. The authors evaluated the tool with two datasets of contracts, one consisting of the Ethereum Virtual Machine (EVM) bytecode of 24,594 smart contracts and another one consisting of 100 Solidity smart contracts. The tool is compared against Oyente [21] and Mythril [6] in terms of detection accuracy and false positives.

Slither [4] is a static analysis tool that can detect various types of contract defects, like shadowing, uninitialized variables, reentrancy, suicidal contracts, locked ether and arbitrary sending of ether; and also defects related with optimization detection, variables that should be declared as constants, and functions that should be declared as externals. The authors have evaluated the tool using a set of 1,000 popular contracts to find out it outperforms Solhint, SmartCheck, and Securify in terms of accuracy and false-positive rates.

The authors in [7] evaluate the bug detection effectiveness of a static analysis tools, namely Oyente, Securify, Mythril, Smartcheck, Manticore, and Slither. The approach is based on the injection of bugs in contracts, based on known bug patterns. The authors inject only the types of defects that the tools claim to be able to detect and use classic metrics like false-positive and false-negative rates to characterize the tools' effectiveness. In [8] the authors evaluate Oyente, Securify, Mythril and Smartcheck using ten contracts and express the tools' effectiveness by performing a Receiver Operating Characteristic (ROC) analysis and also analyze accuracy, revealing differences and gaps in the tools' effectiveness.

A review of nine automated analysis tools for smart contracts is presented in [10]. One of the goals is to analyze the tools' effectiveness, for which the authors use 47,587 ethereum smart contracts (69 manually annotated plus 47,518 which represented, at the time, all the contracts in the ethereum network). There is open space for richer analysis regarding the different outcomes (i.e., true positives, false negatives, and false positives), particularly considering the different types of software faults present in the dataset.

A framework for analyzing and testing smart contracts is presented in [9]. The authors evaluate the proposed tool in perspective with Oyente, Securify, Maian, SmartCheck and Mythril. The evaluation uses 1,838 contracts and 8 faults which are used to produce 12,866 mutated contracts. The different tools are then executed to detect faults and compared via precision and recall. Ye et al. [22] propose a bug benchmark which is then demonstrated by running Oyente and Slither against 1,010 contracts randomly selected from

etherscan.io

In summary, despite the merits of current defect classification schemes used, we find a few limitations with current schemes being used, namely the use of outdated structures, such as in [11], or schemes that tend to only reflect frequent vulnerabilities (e.g., [12]) or require strong adaptations of existing standards [18]. Researches that evaluate verification tools are quite scarce, and in addition to the fast moving pace of the field, we find them either focused on a certain class of tools [4], [5], [7], or using small sets of contracts [8], a low number of faults [9], or lacking deeper analysis regarding tools' capabilities, especially with respect to the different types of software faults that can arise in code [10].

III. Approach and Study Design

This section presents the approach used to evaluate the effectiveness of verification tools in presence of different types of faults in smart contracts. We go through the following steps, described in further detail in the next paragraphs:

1) Definition of a smart contract fault classification;
2) Definition of the set of contracts for the experiments;
3) Selection of contract verification tools;
4) Execution of the tools against the selected contracts;
5) Results analysis, based on a set of metrics of interest.

In *Step 1)* we begin by analyzing smart contract faults and **defining a classification scheme** tailored for this kind of issues, which is structured around the Orthogonal Defect Classification (ODC) defect classification scheme [13] and builds on a preliminary, non-comprehensive, classification used in [23]. We first analyze various sources of contract faults and classifications [2], [11], [12], [14], to find they are generally not comprehensive, being unable to characterize all defects that can appear in smart contracts (at least the known ones). By analyzing multiple classifications and defect sources and finding support in an established scheme like ODC, we aim at providing a framework in which contract defects fit easily.

In *Step 2)*, we **select a set of smart contracts** to be used for the evaluation of verification tools. We are mainly interested in using three types of contracts (as in [11]): i) contracts with known software defects; ii) contracts where known defects have been corrected (and carry no known defects); iii) contracts that have been crafted to mislead verification tools (i.e., contracts that hold a certain vulnerability pattern that is, in practice, not exploitable). Also, we are interested in keeping the set small but, at the same time, holding diverse defects.

We identified three main sources of contracts [11], [14], [15]. As mentioned, we were interested in contracts carrying known defects (named *Set 1 - defective*) and their corrected version (named *Set 2 - fixed*) and also contracts created to mislead tools by holding a certain vulnerability pattern that is, in practice, not exploitable (named *Set 3 - misleading*). The selection was complemented with our own implementation of more contracts (1 from [11], 4 from [14], and 7 from [15]) to allow having at least one corrected version for each defective contract. Thus, we collected 94 defective contracts with 131 known defects, 98 fixed contracts and 30 misleading contracts. The exact numbers per origin can be found in Table I.

TABLE I: The dataset of smart contracts per origin.

Source	Defective	Fixed	Misleading	Total
Zhang et al., 2020	67	70 + 1	30	168
SmartContractSecurity, 2020	20	16 + 4	0	40
Antonopoulos and Wood, 2018	7	0 + 7	0	14
Total Contracts	**94**	**98**	**30**	**222**
Total Known Defects	**131**	**0**	**0**	**131**

Step 3) involves the **selection of contract verification tools**. We aimed at popular tools and also tools of different operational nature. Thus, we selected an abstract interpretation tool (Securify2), a static analysis tool (Slither), and a symbolic execution tool (Mythril), presented in further detail in Table II.

We **execute the tools**, in *Step 4)*, against the whole set of contracts, collect their output and store the relevant information by processing the heterogeneous outputs produced by the tools, mapping the outcome to the analyzed contracts and known faults. The tools are run using default parameters (details available at [17]), thus not being configured to detect any particular type of issues in the contracts.

In *Step 5)* **results are analyzed** and all cases of potential false-positives (i.e., software faults signaled by the tool that in reality do not exist) are manually verified to check if the signaled defect really exists (the contracts may hold unknown vulnerabilities) or not. Overall, we will be examining the tools' overall effectiveness regarding known faults and unknown faults that will be signaled and their complementary abilities in the detection of certain classes of faults present in our fault model. With *Set 1 (defective)* we examine the tools' detection capabilities and the associated effort (i.e., the number of alerts triggered by each tool). In *Set 2 (fixed)* we are firstly interested in examining the true negative rates (the tool signaling a contract as not holding a defect, which should be the case for all contracts in this set considering known defects only). In Set 3 (misleading), we place the tools in presence of contracts that potentially trigger false positives.

IV. Results and Discussion

In this section, we first present the defects/vulnerabilities classification scheme (for the sake of simplicity, we use "defects" in the rest of the document). The scheme was built based on the analysis, from a security perspective, of the data collected from several sources (as discussed in the previous section). We then present the results regarding the evaluation of the effectiveness of the three smart contract verification tools. We begin by overviewing the results and discuss the outcome per type of defect and tool. The section concludes with some results highlights, including clear gaps in the effectiveness of the verification tools.

TABLE II: Tools selected to evaluate the smart contracts.

Tool	Type	Language support	Version Date
Securify v2.0	Abstract interpretation	Solidity >= 0.5.8	13-Apr-20
Slither 0.7.1	Static analysis	Solidity >= 0.4	04-May-21
Mythril v0.22.19	Symbolic execution	Solidity >= 0.4	05-Apr-21

TABLE III: Smart contract defect classification (part 1 of 2).

	Defect Class	Defect Nature	Defect Name	Defect Identifier	Generic/ Specific	Defect Cause	Defect Type	Impact, Threat
1	Assignment	Missing	variable initialization using a value (**MVIV**)	A_MVIV	**G**			
2			variable initialization using an expression (**MVIE**)	A_MVIE	**G**			
3			variable assignment using value (**MVAV**)	A_MVAV	**G**			
4			variable assignment using an expression (**MVAE**)	A_MVAE	**G**			
5			variable auto-increment (**MVAI**)	A_MVAI	**G**			
6			variable Auto-decrement (**MVAD**)	A_MVAD	**G**			
7			Initialization of Storage variables/pointers (Uninitialized Storage Pointer) (**MISP**)	A_MISP	**S**	Language	Storage Access	Uninitialized storage pointer (SWC-109, SP-14), Access of Uninitialized Pointer
8			Initialization of Local Variable (**MILV**)	A_MILV	**S**	Language	Internal control flow	
9			Initialization of State variables (**MISV**)	A_MISV	**S**	Language	Storage Access	DoS
10			Constructor (**MC**)	A_MC	**S**	Model	Authorization	Improper Access Control
11			Compiler Version (**MCV**)	A_MCV	**S**	Language	Compiler	
12		Wrong	arithmetic expression used in assignment (**WVAE**)	A_WVAE	**S**	Language	Arithmetic	Over/underflow
13			miss-by-one value used in variable initialization (**WVIM**)	A_WVIM	**G**			
14			Integer Sign (**WIS**)	A_WIS	**S**	Language	Arithmetic	
15			Integer Truncation (**WIT**)	A_WIT	**S**	Language	Arithmetic	Over/underflow (SWC-101, DASP-3, SP-2), Precision issues (SP-15)
16			value assigned to variable (**WVAV**)	A_WVAV	**G**			
17			value assignment with too many digits (**WVATMD**)	A_WVATMD	**S**	Language	Arithmetic	Maintenance issue
18			use of deprecated build-in symbols (**WUDBS**)	A_WUDBS	**S**	Language	Internal control flow	Hidden Build-in Symbols, DoS, Economic loss
19			build-in symbol name is assigned to variable or function (**WBSAVF**)	A_WBSAVF	**S**	Language	Internal control flow	Hidden Build-in Symbols, DoS, Economic loss
20			value assigned to contract address (**WVAA**)	A_WVAA	**S**	Language	Internal control flow	DoS
21			Constructor name (**WCN**)	A_WCN	**S**	Model	Authorization	Improper Access Control, Constructor name (SWC-118, SP-13)
22			Variable Type (e.g., byte[]) (**WVT**)	A_WVT	**S**	Blockchain	Gas limitations	DoS
23			declaration of invariant state variable (**WDISV**)	A_WDISV	**S**	Language	Internal control flow	Improper Access Control
24			Variable name (Variable Shadowing) (**WVN**)	A_WVN	**S**	Language	Internal control flow	DoS
25			function type variables assignment using arbitrary values (**WFTVA**)	A_WFTVA	**S**	Language	Internal control flow	Arbitrary jump with function type variable (SWC-127)
26		Extraneous	variable assignment using another variable (**EVAV**)	A_EVAV	**G**			
27			function type variables assignment using arbitrary values (**EFTVA**)	A_EFTVA	**S**	Language	Internal control flow	Arbitrary jump with function type variable (SWC-127)
28	Checking	Missing	"if" construct around statement (**MIA**)	CH_MIA	**G**			
29			"require" on transaction sender (**MRTS**)	CH_MRTS	**S**	Model	Authorization	Improper Access Control
30			"require" on input variable(s) (**MRIV**)	CH_MRIV	**S**	Model	Authorization	Improper Input Validation
31			"OR EXPR" in expression used as branch condition (**MLOC**)	CH_MLOC	**G**			
32			"require" OR subexpression on transaction sender (**MROTS**)	CH_MROTS	**S**	Model	Authorization	Improper Access Control
33			"require" OR subexpression on input variable(s) (**MROIV**)	CH_MROIV	**S**	Model	Authorization	Improper Input Validation
34			"if" const. OR subexpression on transaction sender (**MIOTS**)	CH_MIOTS	**S**	Model	Authorization	Improper Access Control
35			"if" construct OR subexpression on input variable(s) (**MIOIV**)	CH_MIOIV	**S**	Model	Authorization	Improper Input Validation
36			"AND EXPR" in expression used as branch condition (**MLAC**)	CH_MLAC	**G**			
37			"require" AND subexpression on transaction sender (**MRATS**)	CH_MRATS	**S**	Model	Authorization	Improper Access Control
38			"if" const. AND subexpression on transaction sender (**MIATS**)	CH_MIATS	**S**	Model	Authorization	Improper Access Control
39			"require" AND subexpression on input variable(s) (**MRAIV**)	CH_MRAIV	**S**	Model	Authorization	Improper Input Validation
40			"if" construct AND subexpression on input variable(s) (**MIAIV**)	CH_MIAIV	**S**	Model	Authorization	Improper Input Validation
41			invariant checking (e.g., current balance) (**MI**)	CH_MI	**S**	Blockchain	Balance transfer	Unexpected Balance transfer
42			check on gas limit (**MCHGL**)	CH_MCHGL	**S**	Blockchain	Gas limitations	DoS
43			check on Return Values for Low Level Calls (**MCHRV**)	CH_MCHRV	**S**	Blockchain	Contract interaction	DoS, Unchecked low-level call (SWC-104, DASP-4, SP-9)
44			check on input address length (**MCHIAL**)	CH_MCHIAL	**S**	Blockchain	Message structure	Short address attack (DASP-9, SP-8)
45			check on External Call (**MCHEC**)	CH_MCHEC	**S**	Blockchain	Contract interaction	DoS with revert (SWC-113, SP-11)
46			check on force to receive balance (**MCHFRB**)	CH_MCHFRB	**S**	Blockchain	Balance transfer	Unexpected Balance transfer
47			check on write to Arbitrary Storage Location (**MCHWASL**)	CH_MCHWASL	**S**	Language	Storage access	Overlap attack (SWC-124)
48			check on arithmetic operation (**MCHAO**)	CH_MCHAO	**S**	Language	Arithmetic	Over/underflow (SWC-101, DASP-3, SP-2), Precision issues (SP-15)
49			check on suicide functionality (**MCHSF**)	CH_MCHSF	**S**	Blockchain	Contract interaction	DoS with selfdestruct (DASP-5), Suicidal contracts (SWC-106), Locked Balance, Unexpected balance transfer
50			check on pre-send balance (**MCHPSB**)	CH_MCHPSB	**S**	Blockchain	Balance transfer	DoS
51			check on Balance Withdrawal (**MCHBW**)	CH_MCHBW	**S**	Model	Authorization	Generous contracts (SWC-105)
52			check on Overpowered owner (**MCHOO**)	CH_MCHOO	**S**	Model	Trust	DoS, Overpowered owner (SP-11 - see 3. Owner operations)
53			check on target address in delegatecall (**MCHTADC**)	CH_MCHTADC	**S**	Language	Storage access	Delegatecall and storage layout (SWC-112, SP-4)
54		Wrong	parenthesis in logical expression used in assignment (**WPLC**)	CH_WPLC	**G**			
55			logical expression used as branch condition (**WLEC**)	CH_WLEC	**G**			
56			"require" for authorization (Authorization through tx.origin) (**WRA**)	CH_WRA	**S**	Model	Authorization	Improper Access Control, Authorization with tx.origin (SWC-115, SP-16)
57			logical expression in "require" over input variable(s) (**WRIV**)	CH_WRIV	**S**	Model	Authorization	Improper Input Validation
58			logical expression in "if" const. over input variable(s) (**WIIV**)	CH_WIIV	**S**	Model	Authorization	Improper Input Validation
59			logical expression in "require" over transaction sender (**WRTS**)	CH_WRTS	**S**	Model	Authorization	Improper Access Control
60			logical expression in "if" const. over transaction sender (**WITS**)	CH_WITS	**S**	Model	Authorization	Improper Access Control
61			check on loop condition (infinite loop) (**WCHLC**)	CH_WCHLC	**S**	Blockchain	Gas limitations	Infinite loops (SWC-129), DoS
62			invariant checking (e.g., current balance) (**WI**)	CH_WI	**S**	Blockchain	Balance transfer	Unexpected Balance transfer
63			check on arithmetic expression in branch condition (**WAEC**)	CH_WAEC	**S**	Language	Arithmetic	Over/underflow (SWC-101, DASP-3, SP-2), Precision issues (SP-15)
64			check on arithmetic operation (WCHAO)	CH_WCHAO	**S**	Language	Arithmetic	Over/underflow (SWC-101, DASP-3, SP-2), Precision issues (SP-15)
65			check on Return Values for Low Level Calls (**WCHRV**)	CH_WCHRV	**S**	Blockchain	Contract interaction	DoS, Unchecked low-level call (SWC-104, DASP-4, SP-9)
66			check on input address length (**WCHIAL**)	CH_WCHIAL	**S**	Blockchain	Message structure	Short address attack (DASP-9, SP-8)
67			check on external Call (**WCHEC**)	CH_WCHEC	**S**	Blockchain	Contract interaction	DoS with revert (SWC-113, SP-11)
68			check on force to receive balance (**WCHFRB**)	CH_WCHFRB	**S**	Blockchain	Balance transfer	Unexpected Balance transfer
69			check on write to Arbitrary Storage Location (**WCHWASL**)	CH_WCHWASL	**S**	Language	Storage access	Overlap attack (SWC-124)
70			check on suicide functionality (**WCHSF**)	CH_WCHSF	**S**	Blockchain	Contract interaction	DoS with selfdestruct (DASP-5), Suicidal contracts (SWC-106), Locked Balance, Unexpected Balance transfer
71			check on Balance Withdrawal (**WCHBW**)	CH_WCHBW	**S**	Model	Authorization	Generous contracts (SWC-105)
72			check on Overpowered owner (**WCHOO**)	CH_WCHOO	**S**	Model	Trust	DoS, Overpowered owner (SP-11 - see 3. Owner operations)
73			check on target address in delegatecall (**WCHTADC**)	CH_WCHTADC	**S**	Language	Storage access	Delegatecall and storage layout (SWC-112, SP-4)
74		Extraneous	check on Overpowered owner (**ECHOO**)	CH_ECHOO	**S**	Model	Trust	DoS, Overpowered owner (SP-11 - see 3. Owner operations)

TABLE IV: Smart contract defect classification (part 2 of 2).

	Defect Class	Defect Nature	Defect Name	Defect Identifier	Generic/ Specific	Defect Cause	Defect Type	Impact, Threat
75	Interface	Missing	return statement (**MRS**)	I_MRS	**G**			
76			"OR sub-expr" in parameters of function call (**MLOP**)	I_MLOP	**G**			
77			"AND sub-expr" in parameters of function call (**MLAP**)	I_MLAP	**G**			
78			visibility modifier of state variables (implicit visibility) (**MVMSV**)	I_MVMSV	**S**	Model	Privacy	Improper Access Control
79			Function Visibility Modifier (MFVM)	I_MFVM	**S**	Model	Authorization	Improper Access Control
80		Wrong	logical expression in parameters of function call (**WLEP**)	I_WLEP	**G**			
81			arithmetic expression in parameters of func. call (**WAEP**)	I_WAEP	**G**			
82			variable used in parameter of function call (**WPFV**)	I_WPFV	**G**			
83			return value (**WRV**)	I_WRV	**G**			
84			assembly code return value in the constructor (**WACRV**)	I_WACRV	**S**	Language	Internal control flow	
85			Character (Right-To-Left-Override control character (U+202E)) within critical information (**WRTLOC**)	I_WRTLOC	**S**	Blockchain	Message structure	DoS, Right-to-Left Override Attack
86			Signature parameter (**WSP**)	I_WSP	**S**	Model	Authorization	Improper Access Control
87			(Nonstandard) Token Interface (**WTI**)	I_WTI	**S**	Model	Authorization	Improper Access Control
88			visibility (public) for private/internal function (**WVPF**)	I_WVPF	**S**	Model	Privacy	Improper Access Control
89			Visibility of Non-public variables through external function	I_WVPVEF	**S**	Model	Privacy	Improper Access Control
90			Interface Implementation (**WII**)	I_WII	**S**	Language	Internal control flow	DoS
91			Sensitive data visibility modifier (**WSDVM**)	I_WSDVM	**S**	Model	Privacy	Improper Access Control
92			Function Visibility Modifier (**WFVM**)	I_WFVM	**S**	Model	Privacy	Improper Access Control
93			parameters in hash function (abi.encode() or abi.encodePacked ()) call (**WPEFC**)	I_WPHFC	**S**	Language	Arithmetic	Hash Collision
94	Algorithm	Missing	function call (**MFC**)	AL_MFC	**G**			
95			call to SafeMath (**MCSM**)	AL_MCSM	**S**	Language	Arithmetic	Over/underflow (SWC-101, DASP-3, SP-2)
96			"if" construct plus statements (**MIFS**)	AL_MIFS	**G**			
97			"if" construct on transaction sender plus statements (**MITSS**)	AL_MITSS	**S**	Model	Authorization	Improper Access Control
98			"if" construct on input variable(s) plus statements (**MIIVS**)	AL_MIIVS	**S**	Model	Authorization	Improper Input Validation
99			"if-else" construct plus statements (**MIES**)	AL_MIES	**G**			
100			"if" const. plus statements plus "else" before statements (**MIEB**)	AL_MIEB	**G**			
101			iteration construct around statements (**MCA**)	AL_MCA	**G**			
102			small and localized part of the algorithm (**MLPA**)	AL_MLPA	**G**			
103			protection against reentrancy (**MPAR**)	AL_MPAR	**S**	Blockchain	Contract interaction	Reentrancy (SWC-107, DASP-1, SP-1)
104			Exception Handling (**MEH**)	AL_MEH	**S**	Language	Internal control flow	DoS
105			continue-statements in do-while-statements (**MCSWS**)	AL_MCSWS	**S**	Language	Internal control flow	DoS
106			protection against re-execution of transaction (**WPARET**)	AL_MPARET	**S**	Language	Internal control flow	Replay Attack
107			protection against gas limit in costly loop (**MPAGLL**)	AL_MPAGLL	**S**	Blockchain	Gas limitations	Infinite loops (SWC-129), DoS
108		Wrong	function called with same parameters (**WFCS**)	AL_WFCS	**G**			
109			function called with different parameters (**WFCD**)	AL_WFCD	**G**			
110			algorithm - code was misplaced (**WALR**)	AL_WALR	**G**			
111			use of require, assert, and revert (**WRAR**)	AL_WRAR	**S**	Language	Internal control flow	Improper Input Validation, DoS
112			(bad) algorithm for generating random number (**WARN**)	AL_WARN	**S**	Blockchain	Block content manipulation	Bad Randomness, Random with blockhash (SWC-120, DASP-6,
113			Protection against reentrancy (**WPAR**)	AL_WPAR	**S**	Blockchain	Contract interaction	Reentrancy (SWC-107, DASP-1, SP-1)
114			Exception Handling (**WEH**)	AL_WEH	**S**	Language	Internal control flow	DoS
115			Dynamic Array Cleanup (**WDAC**)	AL_WDAC	**S**	Language	Internal control flow	DoS
116			continue-statements in do-while-statements (**WCSWS**)	AL_WCSWS	**S**	Blockchain	Gas limitations	Infinite loops (SWC-129), DoS
117			iterating over a dynamically sized data structure (**WIDSDS**)	AL_WIDSDS	**S**	Language	Internal control flow	DoS
118			use of call.value (**WCVALUE**)	AL_WCVALUE	**S**	Blockchain	Contract interaction	Reentrancy (SWC-107, DASP-1, SP-1)
119			use of pull and push in external call (**WPPEC**)	AL_WPPEC	**S**	Language	Internal control flow	DoS
120			Protection against force to receive balance (**WPAFRB**)	AL_WPAFRB	**S**	Language	Internal control flow	DoS
121			protection against re-execution of transaction (**WPARET**)	AL_WPARET	**S**	Language	Internal control flow	Replay Attack
122			protection against gas limit in costly loop (**WPAGLL**)	AL_WPAGLL	**S**	Language	Internal control flow	DoS
123			place of external call (misplaced call to External contract) (**WPEC**)	AL_WPEC	**S**	Blockchain	Contract interaction	Reentrancy (SWC-107, DASP-1, SP-1)
124			use of invariant in loop (**WUIL**)	AL_WUIL	**S**	Blockchain	Gas limitations	DoS
125			Contract logic (e.g., dependent on exact values of the balance of the contract) (**WCL**)	AL_WCL	**S**	Language	Internal control flow	Unexpected Ether transfer, DoS
126			call to wrong hash function (abi.encodePacked()) (**WCEP**)	AL_WCHF	**S**	Language	Arithmetic	Hash Collision
127			Voting logic (**WVL**)	AL_WVL	**S**	Model	Economy	Voting issues
128			economic conjuncture of the token (**WECT**)	AL_WECT	**S**	Model	Economy	Tokenomics issues
129		Extraneous	continue-statements in do-while-statements (**ECSWS**)	AL_ECSWS	**S**	Language	Internal control flow	DoS
130			use of call.value (**ECVALUE**)	AL_ECVALUE	**S**	Blockchain	Contract interaction	Reentrancy (SWC-107, DASP-1, SP-1)
131			operations in fallback function (Complex fallback function) (**WOFF**)	AL_WOFF	**S**	Language	Internal control flow	DoS
132	Function	Missing	Withdraw function (**MWF**)	F_MWF	**S**	Blockchain	Ether transfer	Unexpected Ether transfer, Locked Ether
133			Inheritance (**MINHERITANCE**)	F_MINHERITANCE	**S**	Language	Internal control flow	Missing Functionality
134		Wrong	algorithm - large modifications (**WALL**)	F_WALL	**G**			
135			State modification in constant/view function (**WSMCVF**)	F_WSMCVF	**S**	Model	Authorization	DoS
136			Inheritance and inheritance Order (**WIO**)	F_WIO	**S**	Language	Internal control flow	Multiple inheritance (SWC-125)
137		Extraneous	Inheritance (**EINHERITANCE**)	F_EINHERITANCE	**S**	Language	Internal control flow	Multiple inheritance (SWC-125)
138			function with same signature (**EFWSS**)	F_EFWSS	**S**	Blockchain	Message structure	Signature collisions
139	Timing/serialization	Missing	Race Condition in executing transactions (displacement of transactions) (**MRC**)	T_MRC	**S**	Blockchain	Block content manipulation	Front-running / transaction reordering (SWC-114, DASP-7, SP-10)
140		Wrong	Race Condition in executing transactions (displacement of transactions) (**WRC**)	T_WRC	**S**	Blockchain	Block content manipulation	Front-running / transaction reordering (SWC-114, DASP-7, SP-10)
141		Extraneous	Dependency over mining timestamp to run the transactions (**EDOMT**)	T_EDOMT	**S**	Blockchain	Block content manipulation	Timestamp manipulation (SWC-116, DASP-8, SP-12)

A. Classification Scheme for Smart Contract Defects

Our classification scheme is presented in Table III and Table IV. We identified a total of 141 defects of which 110 defects are smart contract specific (distinguished by *S* in the *Generic/Specific* Column) and 31 defects are general software issues that may also appear in smart contracts (distinguished by *G* in the *Generic/Specific* Column). We identified the *defect class* and *defect nature* attributes based on the ODC classification (respectively named defect type and qualifier in [13]), where the former characterizes the general category of the problem (i.e., Assignment, Checking, Interface, Algorithm, Function, Timing/Serialization) and the latter characterizes the

nature of the defect (i.e., Missing, Wrong, and Extraneous). We then classified the defects in two groups: i) general defects, mostly retrieved from previous work on software faults [16]; ii) smart contract specific defects, identified based on previous work on smart contract faults and security [2], [11], [12]. In the case of smart contract specific defects, we also identified the *cause* of the defect according to [14] (i.e., if it relates to the language, to the model/architecture, or to the blockchain), the *defect type* (e.g., if it relates to gas limitations, balance transfer, or contract interaction), and the *impact/threat* the presence of a certain defect poses, or the impact/threat of being exploited (more details on description of each case can be found in the literature, e.g., [2], [12], [14]).

Figure 1 shows the distribution of the defects based on their cause: *blockchain*, in which defects are caused by the blockchain nature of the system (e.g., block content manipulation); *language*, in which the defects are caused by the insecure use of the language used for implementing contracts (e.g., arithmetic related defects); and *model*, in which defects are caused by mistakes in the model or architecture of the system (e.g., authorization issues). As shown, most of the defects (40%) are associated with the nature of the language. Model-related defects are the second most frequent (32%). The remaining (28%) belong to defects that are caused by the blockchain nature of the system in which the smart contract is running. This implies that, when, for instance, a different language is used to implement smart contracts, some of the 40% of the defects that are related to the language nature (in this study, the smart contracts are implemented in Solidity), may not be valid or may be lacking due to the specific aspects of the language involved.

Regarding the defect type, as we can see in Figure 2, about half of defects belong to the *Internal Control Flow* (25%) and *Authorization* (23%) which are respectively related to the language and model natures. Then, *Contract Interaction* (with 10%) and *Arithmetic* (with 9%) are the next popular defect types, related to the blockchain and language natures.

B. Set 1 - Defective Smart Contracts

The experiments with Set 1 allowed to understand the detection capabilities of the tools. Figure 3 shows the distribution of the known defects in this set of smart contracts. In total, only 57 out of 141 defects of the classification scheme (refer to Tables III and IV) appeared in the defective smart contracts. All 57 defects are smart contract-specific defects.

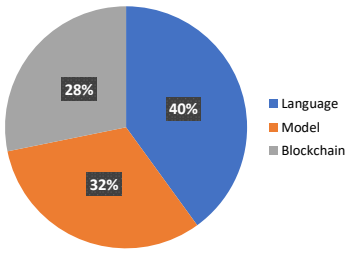

Fig. 1: Distribution of defects based on the defect causes.

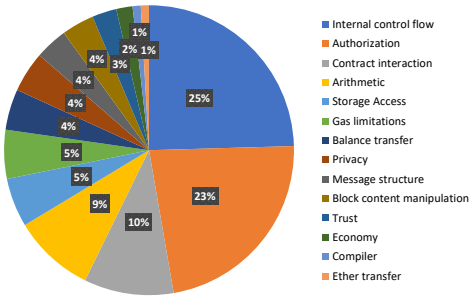

Fig. 2: Distribution of defects based on the defect type.

Among all, `I_WIT - Nonstandard Token Interface` is the most frequent defect followed by `I_MVMSV - Missing visibility modifier of state variables`. Both issues fit the ODC interface class and are model-related. Also, both cause improper access control in smart contracts. The remaining most frequent defects are `A_WUDBS - Wrong use of deprecated build-in symbols`, `A_WCN - Wrong Constructor name`, and `CH_MCHAO - Missing check on arithmetic operation`.

Overall, the tools were able to jointly detect 74 out of the 131 known defects, i.e., 56.5% (considering the combined results). Figure 4 shows a detailed view of the detection capabilities of each tool (in number of detected defects), including the number of defects detected by more than one tool (intersected areas in Figure 4).

As we can see in Figure 4, the tools clearly show complementary detection capabilities. Only 8 defects (out of 74 detected defects) are detected by all three tools (6% of 131). The overlapping regions in Figure 4 account for 28% of the known defects (i.e., 8+8+21 = 37), meaning that less than one-third of the defects are detected by two or more tools. The remaining 37 detected defects (74 - 37 = 37 defects) represent defects detected by a single tool, which again highlights the tools' complementary detection abilities and, at the same time, points out the space for improvement in the tools' detection mechanisms. Table V presents further results regarding the smart contracts from Set 1.

As we can see in Table V, regarding pure detection capability, Slither shows the best results (44%), followed by Securify2 (30%), and Mythril (18%). However, this is achieved at the expense of a relatively high number of false alerts. In fact, the order reverses if we consider the ratio of alerts compared to the correct detection results (i.e., number of alerts divided by the number of correctly detected defects), as, on average, for each detected defect, Mythril produces only 5 alerts,

TABLE V: Results for Set 1 - defective smart contracts.

		Alerts	True Positives		Alerts / TP ratio
			#	%	
Set 1 (defective)	Securify2	433	39	30%	11.1
	Slither	805	57	44%	14.1
	Mythril	104	23	18%	4.5

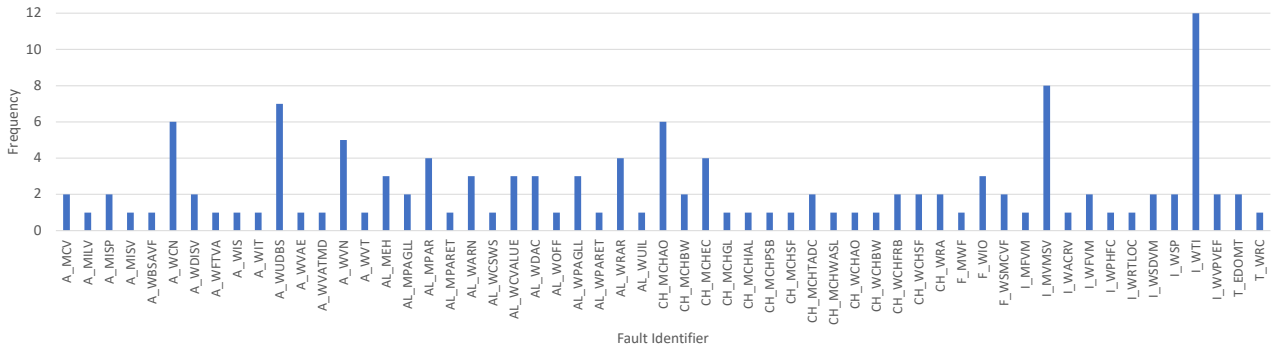

Fig. 3: Distribution of defects in Set 1 - defective smart contracts.

whereas Securify2 produces 11, and Slither produces 14 alerts. This may be important information for application scenarios in which there are constraints on the available resources, including human resources, time, and financial resources, for code review and correction, including the effort associated with identifying false positives.

Figure 5 depicts the capability of the verification tools per type of fault. In general, 21 of 57 types of defects that appeared in the defective smart contracts were not detected by any tool (e.g., AL_WRAR), of which 16 are the least frequent defects (appears only once). The next observation is that the majority of the most frequent defects remained undetected by the tools. For instance, in the case of `I_WTI - Nonstandard Token Interface`, Mythril was not able to detect any of the 12 faulty cases, and Securify2 detected only two of them.

In the case of `I_MVMSV - Missing visibility modifier of state variables`, Slither and Mythril were not able to detect any of the faulty occurrences, and Securify2 only detected 1 case. Despite being known and frequent, this shows that detecting these defects is a challenging task. Among the 57 types of defects, only one of them, namely `F_MWF - Missing Withdraw function`, is detected by all tools, which again shows the different capabilities of the verification tools.

Figure 6 shows the distribution of the detection results in Set 1 according to the defect class (e.g., Assignment or Checking). As shown, the most frequent defects fit in the Assignment class followed by Interface, Algorithm and Checking classes. Timing and Function defects are the least frequent ones. Figure 6 also shows the capabilities of the tools in the detection of these defect classes. Slither is more effective in detecting all defect classes, except for Interface defects. In contrast, Securify2 is more effective in detecting Interface defects. Mythril shows better results than Securify2 when it comes to Checking defects.

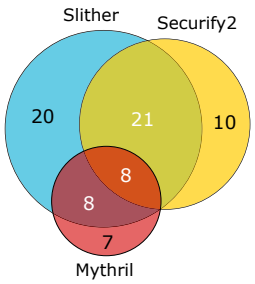

Fig. 4: Tools detection capabilities regarding known vulnerabilities in Set 1.

C. Set 2 - Fixed Smart Contracts

Set 2 is a set of contracts where the known defects are fixed, and it is expected that the tools do not flag the known defects. Corrections were performed in 98 contracts (some of the 94 contracts are corrected in different manners). Figure 7 shows an overview of the tools' performance, with the numbers representing correctly detected fault-free contracts (i.e., the tools correctly report that a given contract does not possess vulnerabilities). Table VI details the individual performance of the tools.

As previously, we can see in both Figure 7 and Table VI that the tools have different and complementary capabilities. Slither generates a large number of alerts and marked all 98 smart contracts as faulty contracts. The other two tools agree that 17% of the contracts (i.e., 17 out of 98) are free of the known defects. In general, Mythril performs better, followed by Securify2, and Slither.

D. Set 3 - Misleading Smart Contracts

Set 3 contains 30 contracts, each of them having some vulnerability pattern that is, however, not exploitable. Figure 8 shows an overview of the tools' different performance, with the numbers representing correctly detected fault-free contracts (i.e., the tools correctly reports that a given contract does not possess vulnerabilities).

Again, we observe in Figure 8 that the tools jointly agree that 43% of the contracts (i.e., 13 out of 30) are free of the known defects. Table VII details the individual performance of

TABLE VI: Results for Set 2 - fixed smart contracts.

		Alerts	True Negatives	
			#	%
Set 2 (98 fixed)	Securify2	426	33	34%
	Slither	712	0	0%
	Mythril	113	52	55%

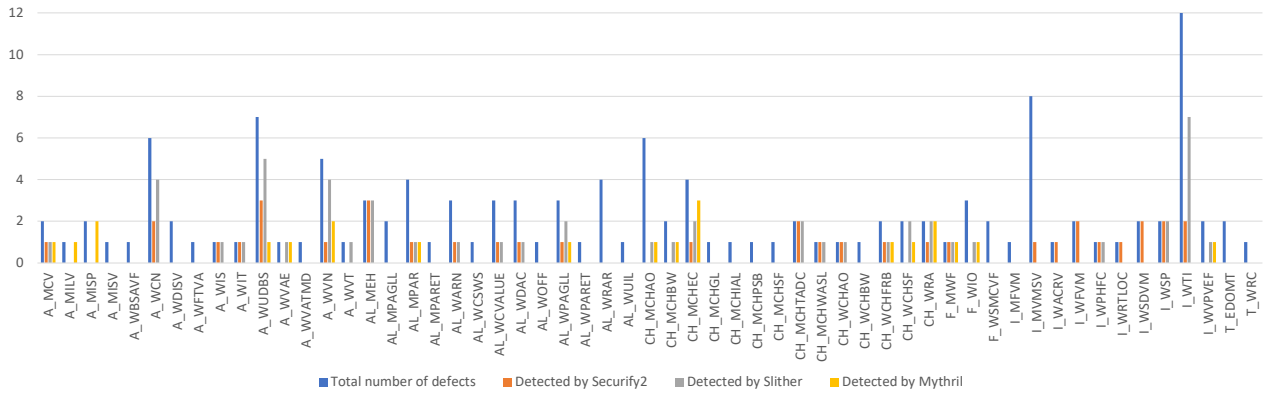

Fig. 5: Tools detection capabilities per defect in Set 1 - defective smart contracts.

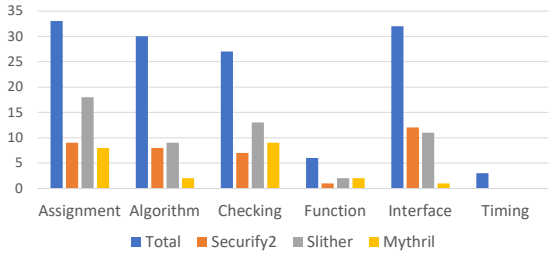

Fig. 6: Tools detection capabilities per defect class in Set 1.

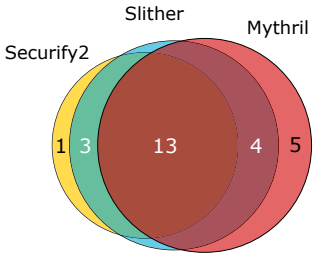

Fig. 8: Correct detection of fault-free contracts in Set 3.

the tools, where we see that Mythril performs better, followed by Slither and Securify2, which tends to be more easily tricked by the misleading contracts.

E. Global Results (Set 1 + Set 2 + Set 3)

Table IX shows global results for the three tools. To be fair about the effectiveness of the verification tools, it is important to use an adequate evaluation criteria. Here, we consider four distinct scenarios where security assurance has different levels of importance, depending on the scenario being evaluated and on the availability of resources to deal with security issues. We select one appropriate criterion for each scenario as a mean to evaluate the tools. The scenarios and associated criteria, follow the proposal in [24] and are as follows:

- **Highly-Critical**: this scenario represents highly business (or safety) critical systems with demanding security requirements (e.g., financial services, medical systems). In this scenario, the detection and removal of software defects, in general, and, more specifically, security vulnerabilities is of high priority (due to the potential high impact of a successful security attack may have on the business or safety of the involved users). Thus, the verification tools should be able to detect the highest number of defective smart contracts, regardless of the amount of possible false positives. In this context, **Recall** [25] is an adequate criterion to evaluate the verification tools, as it characterizes the ratio of defective contracts that are correctly classified independently from the number of false positives.
- **Critical**: this scenario represents systems of critical nature (e.g., e-commerce applications) in which a successful attack tends to lead to the exposure of sensitive data or financial losses. In this context, verification tools should be able to detect the highest number of software defects possible, while trying to avoid reporting too many false positives, as there may be some limits on the use of resources used for correcting and eliminating the defects from the smart contracts. In this case, **Bookmaker Informedness** [25] is

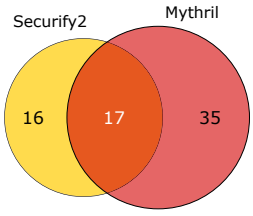

Fig. 7: Correct detection of fault-free contracts in Set 2.

TABLE VII: Results for Set 3 - misleading smart contracts).

		Alerts	True Negatives	
			#	%
Set 3 (30 misleading)	Securify2	126	17	57%
	Slither	124	20	67%
	Mythril	21	22	73%

TABLE VIII: Application scenarios and criteria.

Scenario	Criterion	Formula
Highly-Critical	Recall	$\frac{TP}{P} = \frac{TP}{TP+FN}$
Critical	Bookmaker Informedness	$\frac{TP}{P} - \frac{FP}{N} = \frac{TP}{TP+FN} - \frac{FP}{TN+FP}$
Low-Critical	F-Measure	$2 * \frac{precision * recall}{precision + recall} = \frac{2 * TP}{2 * TP + FN + FP}$
Non-Critical	Markedness	$Precision + inverse\ precision - 1 = \frac{TP}{TP+FP} + \frac{TN}{FN+TN} - 1$

an appropriate criterion, as it associates a high importance to true positive rate with moderate penalization for tools reporting high numbers of false positives.

- **Low-Critical**: this scenario refers to blockchain systems that are less critical and less exposed to attacks. In such scenarios, the resources for detecting and removing bugs are limited. Thus, both detecting and eliminating the highest number of defects and spending less resources for analyzing false positives have similar priorities. This is adequately reflected by **F-Measure** [25], which combines precision and recall.
- **Non-Critical**: this scenario refers to the case of systems that are non-critical (i.e., from a security perspective). Thus, there is a greater concern with the number of false positives due to resource restrictions, although there is still interest in detecting and removing defects. **Markedness** [25] is an adequate criterion in this context, as it rewards low false positive rates and, at the same time, takes true positives into account.

Table VIII presents the identified criteria and respective formulas. In the formulas visible in Table VIII, True Positive (TP) refers to the number of defective contracts that are correctly identified, True Negative (TN) represents the number of non-defective contracts that are correctly identified, False Positive (FP) represents the number of non-defective contracts that are wrongly identified as defective and False Negative (FN) represents the number of defective contracts that are wrongly identified as non-defective.

As we can see in Table IX, we calculated the four criteria for four distinct scenarios. The results show that none of the verification tools analyzed in this study are suitable for any of the scenarios due to the low number of true positives (defective contracts that are correctly identified as defective) and to the high number of false positives, hence the low and negative metrics' values. However, among the three tools, Slither generally shows better results and has a recall over 50 percent (i.e., 0.61), but with the cost of a very high false-positive rate.

F. Discussion of The Results

The main findings of this work are as follows:

- Smart contracts defects are associated with diverse causes, namely language, model and blockchain natures. Thus, at the time of writing, there is no unique classification scheme for all types of smart contracts implemented in different programming languages and running on distinct blockchain systems. Still, the scheme presented in this work includes a high number of defects that are common in smart contracts and reflect the context of a very popular programming language used for writing smart contracts – Solidity.
- Most of the defects identified in the classification scheme are related to the ODC Interface class of defects. However, Assignment defects are more frequent in the faulty smart contracts, followed by Interface and Checking classes.
- The effectiveness of the smart contract verification tools is quite low due to the low number of true positives and the high number of false positives generated. In practice and from the security perspective, none of the tools analyzed in this work is suitable to be used in the identified scenarios.
- In general, Slither showed better results in terms of recall, but at the cost of a very high false-positive rate.
- In general, the smart contract verification tools show complementary capabilities. Thus, creating a tool ensemble that makes effective use of the different capabilities involved is a possible path towards higher performance.

V. Threats to Validity

In this section, we present threats to the validity of this work and discuss mitigation strategies. We begin by mentioning that *selecting a specific software defect classification scheme* to fit the smart contract defects may result in providing inaccurate views of the tools' capabilities, if, for instance, the definition of the defect classes are imbalanced, with some aggregating a large number of different types of defects. Also, the classification may not clearly represent certain types of defects, leading to the misclassification of certain bugs and aggravating the view of the tools' effectiveness. To mitigate this issue, we selected a widely used and very popular defect classification scheme (i.e., ODC) in favor of schemes whose application revealed issues in related work [11], [18] or are just a partial representation of contract defects, e.g., [12].

The *number of software defects or contracts used* may result in a inaccurate view of the tools' effectiveness (e.g., due to missing diversity of vulnerabilities). We tried to diminish this issue by, based on ODC, selecting different types of defects, while trying to keep the dataset size relatively small (as a way to decrease the manual effort involved).

We performed a *manual classification of the identified defects* using ODC. As a manual step, this may introduce some errors, as the process was initially carried out by an Early Stage Researcher. To attenuate this issue, an Experienced Researcher double-checked the classification. Divergencies were discussed and eliminated.

The *selected tools* may not provide a proper vision of smart contract verification tools as the set is relatively small. Still, we selected tools that frequently appear cited in the literature and appear to be actively developed. Finally, the *results were manually analyzed* initially by an Early Stage Researcher, still this process was accompanied by an Experienced Researcher

TABLE IX: Global results.

		Alerts	TP	TN	FP	FN	Recall	Informedness	F-Measure	Markedness
All Smart Contracts (Set1+Set2+Set3)	Securify2	985	39	50	78	55	0,41	-0,19	0,37	-0,19
	Slither	1641	57	20	108	37	0,61	-0,24	0,44	-0,30
	Mythril	238	23	74	54	71	0,24	-0,18	0,27	-0,19

which also verified a subset of the results. Any classification conflicts were discussed and resolved.

VI. Conclusion

In this paper, we carry out an empirical study to understand the effectiveness of three popular smart contract verification tools. We begin by defining a framework, structured around the Orthogonal Defect Classification, in which we fit known smart contract defects. We then use three sources of smart contracts [11], [14], [15] and define an heterogeneous set of contracts holding different types of software defects and which we use against Mythril [6], Securify2 [5], and Slither [4].

Results show the relatively low detection effectiveness of the tools, but especially show their complementarity regarding the detection of different types of faults. The tools also allowed us to identify new cases of faults in the analyzed smart contracts, although this is carried out at the expense of manually analyzing alerts, which are also produced in different orders of magnitude by the tools. In future work, we intend to implement a smart contract fault injector that will allow generating realistic faulty contracts, which will be used in a more extensive tool evaluation.

Acknowledgements

This work has been supported by national funds through the FCT - Foundation for Science and Technology, I.P., within the scope of the project CISUC - UID/CEC/00326/2020 and by European Social Fund, through the Regional Operational Program Centro 2020; and by the TalkConnect project "Voice Architecture over Distributed Network" (reference: POCI-01-0247-FEDER-039676), co-financed by the European Regional Development Fund, through Portugal 2020 (PT2020), and by the Competitiveness and Internationalization Operational Programme (COMPETE 2020).

References

[1] E. G. Sirer, "Thoughts on The DAO Hack," 2016. [Online]. Available: https://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/

[2] A. Manning, "Solidity Security: Comprehensive list of known attack vectors and common anti-patterns." [Online]. Available: https://blog.sigmaprime.io/solidity-security.html

[3] J. D. Arthur and J. B. Dabney, "Applying standard independent verification and validation (IV V) techniques within an Agile framework: Is there a compatibility issue?" in *2017 Annual IEEE International Systems Conference (SysCon)*, Apr. 2017, pp. 1–5.

[4] J. Feist, G. Grieco, and A. Groce, "Slither: A Static Analysis Framework for Smart Contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, May 2019, pp. 8–15.

[5] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical Security Analysis of Smart Contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 67–82.

[6] Consensys, "Mythril," May 2021. [Online]. Available: https://github.com/ConsenSys/mythril

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

[8] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh, "Empirical vulnerability analysis of automated smart contracts security testing on blockchains," in *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '18. USA: IBM Corp., Oct. 2018, pp. 103–113.

[9] S. Akca, A. Rajan, and C. Peng, "SolAnalyser: A Framework for Analysing and Testing Smart Contracts," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, Dec. 2019, pp. 482–489.

[10] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 530–541.

[11] P. Zhang, F. Xiao, and X. Luo, "A Framework and DataSet for Bugs in Ethereum Smart Contracts," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2020, pp. 139–150.

[12] NCCGroup, "Decentralized Application Security Project (DASP) Top10," 2021. [Online]. Available: https://dasp.co/

[13] IBM. (2013, Sep.) Orthogonal Defect Classification v 5.2 for Software Design and Code. [Online]. Available: https://researcher.watson.ibm.com/researcher/files/us-pasanth/ODC-5-2.pdf

[14] SmartContractSecurity, "Smart Contract Weakness Classification (SWC) and Test Cases," 2020. [Online]. Available: http://swcregistry.io/

[15] A. M. Antonopoulos and G. Wood, *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, Inc., Nov. 2018.

[16] J. Duraes and H. Madeira, "Emulation of Software Faults: A Field Data Study and a Practical Approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 849–867, Nov. 2006.

[17] B. Dias, N. Ivaki, and N. Laranjeiro, "An empirical evaluation of the effectiveness of smart contract verification tools - supplemental material," Sep. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5512154

[18] W. Dingman, A. Cohen, N. Ferrara, A. Lynch, P. Jasinski, P. E. Black, and L. Deng, "Defects and Vulnerabilities in Smart Contracts, a Classification using the NIST Bugs Framework," *International Journal of Networked and Distributed Computing*, vol. 7, no. 3, pp. 121–132, Jul. 2019.

[19] H. Poston, "Mapping the OWASP Top Ten to Blockchain," *Procedia Computer Science*, vol. 177, pp. 613–617, Jan. 2020.

[20] OWASP, "OWASP Top Ten," May 2021. [Online]. Available: https://github.com/OWASP/Top10

[21] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 254–269.

[22] J. Ye, M. Ma, T. Peng, Y. Peng, and Y. Xue, "Towards Automated Generation of Bug Benchmark for Smart Contracts," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Apr. 2019, pp. 184–187.

[23] A. Hajdu, N. Ivaki, I. Kocsis, A. Klenik, L. Gonczy, N. Laranjeiro, H. Madeira, and A. Pataricza, "Using Fault Injection to Assess Blockchain Systems in Presence of Faulty Smart Contracts," *IEEE Access*, vol. 8, pp. 190 760–190 783, 2020.

[24] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira, "Vulnerable code detection using software metrics and machine learning," *IEEE Access*, vol. 8, pp. 219 174–219 198, 2020.

[25] N. Antunes and M. Vieira, "On the metrics for benchmarking vulnerability detection tools," in *2015 45th Annual IEEE/IFIP international conference on dependable systems and networks*. IEEE, 2015, pp. 505–516.