Journal:

Journal of Systems and Software

Title:

Security Evaluation and Improvement of Solidity Smart Contracts

Corresponding author:

Mirko Staderini, Department of Mathematics and Informatics, University of Florence, Florence – Italy

E-mail - present address: mirko.staderini@unifi.it

E-mail - permanent address: mstaderini@gmail.com

Biography:

Mirko Staderini is a Ph.D. candidate in computer science at the University of Florence, Italy. His research activities include research and experimentation of dependable and secure systems. His research crosses the domains of safety, security, and software engineering, focusing on Blockchain.

Other Authors:

Andràs Pataricza, Department of Measurement and Information Systems, Budapest University of Technology and Economics, Budapest – Hungary

E-mail: pataricza.andras@vik.bme.hu

Biography:

András Pataricza received the D.Sc. degree from the Hungarian Academy of Sciences (MTA) and Dr.Habil. degree from BME. He founded the Fault-Tolerant Systems Research Group in 1994 at the Department of Measurements and Information Systems. He received multiple recognition awards from different scientific and industrial organizations, like the Officers Cross of Merit from the President of Hungary, the Academic Award of the MTA, and six IBM Faculty Awards. He has published over 190 articles in international journals and conferences in dependable computing, cyber-physical systems, and model-driven engineering. He has been chairing the program committee in several international conferences."

Andrea Bondavalli, Department of Mathematics and Informatics, University of Florence, Florence – Italy

E-mail: andrea.bondavalli.unifi.it

Biography:

Andrea Bondavalli is currently a Full Professor of computer science at the University of Florence. His research interests include designing and evaluating resilient and secure systems and infrastructures. In particular, he has been working on safety, security, fault tolerance, and evaluating attributes such as reliability, availability, and performability. His scientific activities resulted in more than 220 articles appearing in international journals and conferences. He led various national and European projects and has chaired the steering and program committees in several international conferences. He is a member of the IFIP W. G. 10.4 Working Group on "Dependable Computing and Fault-Tolerance.".

Contents lists available at ScienceDirect

The Journal of Systems and Software

Journal Homepage: www.elsevier.com/locate/jss

Security Evaluation and Improvement of Solidity Smart Contracts

Mirko Staderini [a,*], ***Andràs Pataricza*** [b], ***Andrea Bondavalli*** [a]

[a] *Department of Mathematics and Informatics, University of Florence, Florence – Italy*
[b] *Department of Measurement and Information Systems, Budapest University of Technology and Economics,Budapest - Hungary*

ARTICLE INFO

Article history:
Received: xx Month 2022

Keywords:
Smart Contract Security
Smart Contract Vulnerability
Static Analysys tools
Performance
Coverage
Criticality analysis

ABSTRACT

Smart contracts are the major innovation of the second generation of Blockchain. The automated execution of a faulty contract endangers applications as they are immutable after storing them on a Blockchain. Attackers can exploit vulnerabilities originating in hidden weaknesses. Impacts may be critical as a single vulnerability can cause severe (financial) losses. Static analysis is an efficient and widespread methodology to detect and remove weaknesses from a code. This work focuses on the weakness detection capabilities of a collection of static analysers for Solidity, the primary programming language used to develop smart contracts for Ethereum (one of the most popular platforms for smart contracts). The paper classifies a set of Solidity vulnerabilities (33) to a language-independent taxonomy based on the Common Weakness Enumeration (CWE). This way, it provides a fault model for Solidity (language release independent). It performs then a comparative analysis of the capabilities of some selected static analysis tools to detect weakness originating vulnerabilities on a representative set of smart contracts. This evaluation (i) identifies the (un)covered vulnerabilities and the related classes; (ii) quantifies the testing efficiency of the individual static analysers; (iii) improves the coverage by combining multiple tools. In fact, considering tools as black boxes, we quantitatively determine that using a single tool (even the best one) is not optimising security; combining four tools allows to achieve a coverage of 0.9. (iv) Finally, we analyse and prioritise the uncovered vulnerabilities. This way, we provide an indicator for developers to increase the tool impact in smart contract security by identifying the most severe vulnerabilities not covered by the static analysis tools.

1 Introduction

Blockchain [1] promises an out-of-the-box solution to improve the security of critical distributed systems despite not being a panacea [2]. Its core idea is to keep the individual copies of the joint system state in complete synchrony at each cooperating partner by executing the same code (smart contracts [3]) over identical data.

Blockchain protects smart contracts, data, and transaction logs by a strong hash encoding, thus ensuring their immutability and non-repudiability. However, design and coding faults and *weaknesses* in the smart contracts implementing the particular application can still result in exploitable *vulnerabilities* to malicious attacks despite the well-designed run-time environment. The need for an end-to-end assurance of critical applications has been and is considered vital and, consequently, it led to developing several automated vulnerability detection tools.

Ethereum [4] is one of the most widely used platforms for smart contracts. Solidity [5] is the primary (and Turing-complete) programming language that targets the development of Ethereum smart contracts. One of the major open problems related to the Ethereum Blockchain is the insufficiency of quality assurance. Vulnerability records are still sparse due to Solidity technology's novelty and rapid evolution. Consequently, the existing smart

* Corresponding author, e-mail address (present/permanent): mirko.staderini@unifi.it, mirko.staderini@unifi.it (M.Staderini)
Other authors, e-mail addresses: pataricza.andras@vik.bme.hu (A. Pataricza), andrea.bondavalli@unifi.it (A. Bondavalli).

contract checking tools are still immature. On the other hand, Solidity is just another new programming language reusing its central notions from traditional ones extended by Ethereum-specific elements. This way, the most promising way to create a quality assurance process for Solidity is adapting existing technologies to the peculiarities of Ethereum and, in particular, Solidity.

The paper assesses the primary resources of a typical software checking process. (i) It empirically derives a model for characteristic weaknesses from a set of vulnerable smart contracts and cross-compares it with that elaborated for conventional programming languages. (ii) It evaluates the efficiency of one of the most widely used checking technologies, static analysis for Solidity.

2 Background and contributions

2.1 Ethereum and Solidity

The *Ethereum Virtual Machine* (EVM) is the execution environment of Ethereum: each execution changes the EVM *state* [6]. Ethereum has two kinds of accounts: *externally owned accounts* and *contract accounts*; both have a *balance* field in *Ether* (the native currency of Ethereum). The first kind of account represents a user account, controlled by its private key; the second is a smart contract account, and its code controls it. A user can send a transaction (signed data package) to other accounts: if the receiver is a contract, it activates its code executing it into the EVM. Other contracts can trigger the code execution of a contract by messages (function calls). Blocks of the Blockchain contain all transactions and the related EVM state.

The *gas* represents *fees* to be paid for computations in Ethereum. Executing a transaction requires *computational steps* and then fees. Every transaction contains a *recipient*, a *sender* (identified by a signature), a *startgas* (maximum number of computational steps), and a *gasprice* (the fee the sender pays for each unit of gas). The product of startgas by gasprice is the *maximum* fee (in the units of Ether) paid to the *miner* processing the transaction. If the transaction terminates successfully, the miner returns unused gas to the sender.

The primary high-level language used to develop Ethereum smart contracts is *Solidity* [5]. C++, Python, and Javascript influenced Solidity. It has a similar programming structure to traditional languages, such as several types of variables, branching instructions, and assertions.

The core of a smart contract consists of one or more logic contracts (*contract* keyword) and optionally libraries and interfaces containing state variables and functions. Functions can execute instructions, interact with other contracts and modify state variables. Each change of a state variable is saved permanently into the Blockchain. Moreover, a *modifier* changes the visibility of a function and its capability to receive Ether. Besides, functions receive the detail of the transaction. Once deployed into the Blockchain, the contract gets its address, a *constructor* (a special function) initialises its variables, and its code becomes immutable.

2.2 Quality assurance

Quality assurance (QA) is one of the most tradition-richest fields in software engineering. The ISO/IEC 25000 family of standards defines system and software quality requirements and evaluation at the behavioural level, including extra-functional properties. The new ISO/IEC 5055:2021 standard [7] measures source code quality in an automated way based on an estimation of the complexity of the code under test and an empirical set of antipatterns, i.e., typical weaknesses causing failures.

This way, our basic assumption is that most weaknesses and resulting vulnerabilities in Solidity are similar to those in conventional programming languages, potentially appearing in a specific form.

Software security engineering has solid empirical foundations from a well-organised and maintained vulnerability data acquisition, abstraction, and generalisation process. The *Common Vulnerabilities and Exposures* (CVE) database [8] collects, defines, and catalogues publicly disclosed cybersecurity vulnerabilities, i.e., weaknesses in software (and hardware) components that, when exploited, spoil the security of the system. The *Common Weakness Enumeration* (CWE) classifies weaknesses as root causes of vulnerabilities into a hierarchical taxonomy; furthermore, each CWE list item highlights the mode of introduction, expected consequences, and potential mitigations.

The hierarchy starting at the *Variant* level and continuing with the *Base* (typically related to a particular product, language, or technology) gradually eliminates the implementation details. The highly abstract top two categories of weaknesses in the hierarchy (*Class* and *Pillar*) are already independent of any specific language or technology. At the topmost level of abstraction, *Pillar* is a concept used to group weaknesses that share common characteristics. The notion of *Class* is abstract enough to be implementation independent but close enough to define functional and/or structural antipatterns leading to weaknesses.

CVE currently has more than 160000 entries, but only a few hundred are related to Blockchain technologies. However, the close relation of Solidity to traditional programming languages justifies the reuse of CWE classes for faithful fault modelling.

The CWE guides classification of vulnerabilities into the US government repository entitled *National Vulnerability Database* (NVD) [9] used for vulnerability management, security measurement, and compliance in a general technology context.

While CWE has a primary security focus, it is appropriate to describe weaknesses considering other extra-functional properties. For instance, as mentioned before, ISO/IEC 5055:2021 measures the reliability, performance, and maintenance of a code using CWE patterns.

The paper focuses on the aspect of security. Accordingly, we refer to *vulnerabilities* as exploitable *weaknesses*.

2.3 Static analysis

Static analysis (SA) is one of the most significant and widely used types of code analysis. It inspects the code without executing it. At first, it extracts an abstract model of the code under evaluation (typically an AST = *abstract syntax tree* or CFG = *control flow graph*). It searches potential vulnerabilities in the code over the model by pattern matching for weaknesses (*antipatterns*). Its low effort demand compensates for its incomplete (but for most applications still sufficient) fault coverage [10]. Several *static analysis (SA) tools* developed in the last years focus explicitly on Solidity smart contracts vulnerability detection.

2.4 Approach

Traditional languages (e.g., C, Java) have a long tradition of using SA. The broad spectrum of available tools raised the need to evaluate and compare them in a benchmark-styled way.

The main objective of end-users in using testing technologies is to assure a good quality of the code and the productivity of the development process. At the same time, estimation of the

insufficiencies is a primary input to the tool developers in their product development strategy.

Guaranteed quality necessitates a high probability of detecting faults, while productivity needs effective diagnosis and localisation to support debugging.

Measures of the detection effectiveness of an SA tool primarily address fault coverage, i.e., the ratio of fault kinds covered vs all faults anticipated. The minimisation of security risks additionally considers the frequency of occurrence and the severity of impacts of individual vulnerabilities in a more fine granular calculation of the fault coverage.

Creating the benchmark input set for software testing tools underlies several requirements. The programs consisting of the input benchmarking set have to represent the anticipated faults rooting in and covering observed vulnerabilities. This way, the set of tool testing input programs results from extensive vulnerabilities collection, root-cause analysis, categorisation, and potentially generalisation (e.g., mapping to CE) process. Evaluating the vulnerability and diagnosis capabilities of the tools requires the avoidance of benchmark test programs incorporating multiple vulnerabilities to avoid potential interferences between them.

Note that the statistically valid estimation of the frequency of occurrence needs a sufficiently large scale database. The assessment of the severity characteristics requires expert analysis of the potential impacts of a vulnerability.

One of the first examples to standardise the evaluation of SA tools was the NIST Software Assurance Reference Dataset Project covering, resulting in Juliet Test Suites for the most widely used programming languages, like Java, C/C++, and C# [11].

The main resulting vital points are:

- It is possible to identify the *trigger* of the corresponding vulnerabilities and weaknesses, which permits the study of the coverage.
- Each language-specific Juliet Test Suite can rely on an extensive, well-organised historical vulnerabilities database. Juliet anticipates a set of *Base* or *beyond the Base level* weaknesses from the CWE hierarchy as the source of vulnerabilities and covers them by test programs. Using such low-level weaknesses as a basis assures a close correlation between the tests and vulnerabilities. At the same time, the CWE-benchmark mapping delivers the ground truth by construction.
- The availability of a considerable number of (patterns) candidates permits a selective choice for inclusion into the benchmark to reduce false diagnosis due to multi-vulnerability interferences without compromising the fault coverage.

While the sound engineering principles of Juliet lead to a wide practical use for traditional programming languages, their adaptation to evolving languages faces severe difficulties due to the lack of sufficiently extensive experience.

Solidity, as already stated, is a new and evolving technology with changes in the language specification, as typical for technologies in their early lifecycle. According to the novelty of Solidity, only a limited set of well-documented and analysed faulty patterns is available. Moreover, the relatively rapid evolution of the language definitions eliminates the most prevalent ones by improving the safety of the language constructs offered to the programmer. No similar deep analysis of the remaining patterns has happened yet, as for the repositories of vulnerable code fragments in traditional languages. Accordingly, there is no clear ground truth related to faults and selectivity.

In Solidity, it is possible having very similar patterns with different impacts. The lack of an extensive base confines the options of composing the benchmark set, especially to comply with the requirement of selectivity. Accordingly, we used the *natural language* description of vulnerabilities in the selection process.

At the same time, in the case of Solidity, our experience indicates that the current solutions are not extraordinarily sophisticated and well-elaborated. Thus, we use a top-down approach for the reasons listed above and contrary to Juliet. We generalise typical patterns into high-level categories by referring to *Classes* or *Pillars* instead of going at the lowest level of the CWE hierarchy *(*using *Bases or Variants).* Using large aggregate types of weaknesses avoids statistically unjustified partitioning of a relatively sparse dataset. In addition, we also have to create the ground truth due to the inadequate evaluation and documentation of vulnerability patterns.

Our patterns for evaluation can not cover all the potential patterns; however, we provide a representative model of Solidity in terms of observed vulnerabilities and weaknesses extracted from them. That means we can not guarantee that each pattern base is covered, but we offer a rough granular characterisation of the individual SA tools.

2.5 Contribution and paper structure

This paper addresses the security evaluation of Solidity smart contracts based on the empirical assessment of efficiency of the SA-based checking. The analysis deals with Solidity Version 0.5 and up due to the incompatibility of new and previous versions (the latest release is 0.8).

We address the following main research question: *How can we evaluate and improve the security of smart contracts by using SA to detect the most relevant vulnerabilities?* In particular, we investigate the following sub-questions:

RQ1: *What are the classes of vulnerabilities-related weaknesses that remain undetected by the different SA tools?*

RQ2: *What are the individual SA tools' checking performance metrics (coverage, diagnostic accuracy)?*

RQ3: *Can one improve the checking performance by combining different SA tools?*

RQ4: *Which are the most critical vulnerabilities which remain undetected by SA tools?*

The remainder of the paper proceeds as follows.

- Section 3 analyses the *available collections of Solidity vulnerabilities and maps the most important ones to the CWE taxonomy*, following the methodology proposed by [12].
- Section 4 focuses on the detection and diagnosis *capabilities* of a collection of open-source SA tools. Documentation analysis extended with *reverse engineering* of the internal checking rules allows identifying the target set of weaknesses for each tool (and therefore the CWE classes not targeted by the particular SA tool) (RQ1).
- Section 5 first describes the extraction of a *random subset of smart contracts* from the Etherscan public repository to construct our *reference set,* which will serve as an analysis target in the remainder. Moreover, manual inspection and labelling of a subset of contracts (our pilot set) allow us to

determine the ***ground truth***, identifying the *type* and *location* of their vulnerabilities.

- Section 6 focuses on computing *basic statistical metrics* for comparing the diagnosis capabilities of the different SA tools to quantify: i) how well they perform on their anticipated weaknesses; ii) to benchmark the SA tools detection performance when exposed to a set of smart contracts (RQ2).
- Section 7 discusses and experiments on the feasibility of improving detection coverage by using a combination of available tools (RQ3).
- Section 8 analyses the uncovered vulnerabilities. The aim is to provide an indicator for developers by identifying and prioritising the most severe vulnerabilities not covered by the static analysis tools.
- The paper concludes by discussing the limitations of the analysis (Section 9), summarising related works (Section 10), and providing the conclusion (Section 11).

3 Fault modelling for Solidity

This section identifies and systemises the vulnerabilities addressed in this work. The focus is on vulnerabilities originating in the development process of smart contracts. Accordingly, the underlying run-time platform is considered only as a potential error propagation path. Existing surveys (e.g., [13]) and sites provide lists of vulnerabilities, which, unfortunately, are updated overly frequently due to the relative novelty of the technology.

To provide a list of vulnerabilities, we ran a Google Scholar search using the keywords "Ethereum survey," "smart contracts analysis," "smart contracts vulnerabilities." Additionally, we consulted several repositories to create a unified list of candidate vulnerabilities (*Smart Contract Weakness Classification and Test Case* (SWC) registry [14], Solidity documentation [5], Ethereum Smart Contracts, and Best Practices [15], and other referenced GitHub lists). We then discarded from this list all the historical vulnerabilities already resolved at the language level from the Solidity release 0.5 onwards. This cleansing utilised the standard Ethereum improvement guidance in *Ethereum Improvement Proposals* (EIPs). Here the transition to a new version mitigated or eliminated some vulnerabilities.

Finally, grouping vulnerabilities with similar or overlapped definitions resulted in a list of 33 items grouped in a language-independent classification. This classification uses general abstract classes from a subset of the CWE-1000 research concept [16] (based on abstractions of software behaviour).

Starting from the identification of the lowest level of the hierarchy that best fits each vulnerability and proceeding bottom-up (as proposed in [12]), we find *Classes* or *Pillars* (in the *Pillar* sub-hierarchy) that group vulnerabilities with a similar behaviour: then we select them as our classification categories.

Consider two vulnerabilities as examples:

1) *Gasless send* (Gs): it happens when a call invocation provides a limited quantity of gas to the callee, and the gas consumption for executing an operation exceeds the provided amount [17].

2) *DoS costly Patterns and Loops* (CPL): a DoS situation can happen if the gas needed to complete an execution exceeds the gas block limit [13]. A potential cause is using an unbounded operation (e.g., loops that depend on a function parameter).

Our target is to identify the CWE *Base* or *Class*-level weakness corresponding to the best abstraction. As CWE is only a semi-formal taxonomy without formal semantics, mapping Solidity vulnerabilities into CWE is manual. After identifying the main characteristics of the vulnerability (e.g., the software's resources' exhaustion), an abstract keyword and synonym search in the CWE list identifies the candidate shortlist classes. An in-depth review of every element in this list leads to selecting the specific Class or Base to represent the vulnerability. This process follows the *criteria for the best match* in [18]. In the case of the example, class CWE-400 covers vulnerabilities.

The classification of 33 vulnerabilities (Table 1) contains the acronym (Acr.), the full name, the reason for classification, the CWE category (specifying *Class* or *Pillar*), and finally, a short description.

As the next step, we check that the similarity between Solidity and conventional programming languages manifests in their respective fault model. We compare our proposed model for Solidity (10 CWE-IDs) and that of the ISO/IEC 5055:2021 standard (71 CWE-IDs) for conventional languages.

Note that the comparison of the fault models should cover both identities and similarities of the weaknesses in the two fault models. For instance, two weaknesses sharing a joint abstract ancestor in the CWE hierarchy indicate a common root cause manifested in different forms due to the peculiarities of the different programming languages.

12 CWE-IDs in the ISO list are irrelevant as Solidity has no similar language constructs. 7 ISO CWE-IDs related to software obsolescence are irrelevant for our purposes. 20 ISO CWE-IDs are descendants of our Solidity model. One CWE-ID is identical in the ISO and Solidity lists. Two Solidity and ISO CWE-IDs have a joint ancestor in the CWE hierarchy. 29 ISO CWE-IDs are derivatives of the family containing our Solidity CWE-IDs. The model for *Solidity* includes *3 Solidity-specific CWE-IDs that extend the list* of the ISO standard:

- *CWE-330 (Use of Insufficiently Random Values)* is related to generating random numbers whose seed comes from a value stored in the Blockchain.
- *CWE-345 (Insufficient Verification of Data Authenticity)*, concerning the use of signature and invalid data.
- *CWE-284 (Improper Access Control),* related to improper authorisations or controls that permit unauthorised operations and Ether manipulations.

The fault models for the general-purpose elements in conventional languages (ISO standard) and Solidity are similar, while in addition, Solidity has some Ethereum-specific faults. If the Solidity-relevant fault classes are identical to or descendants of the ISO standards, then the algorithms elaborated for traditional languages are reusable. Otherwise, new or extended algorithms are needed.

4 Static analysers

This section describes the characteristics of SA tools, the tools selection process, and a qualitative analysis identifying vulnerabilities not targeted for detection by the individual tools (**RQ1**).

- distinguish here two possibilities:

Table 1
Mapping between vulnerabilities and CWE based classification.

Vulnerabilities		Classification		
Acr.	***Name***	***Reason for classification***	***CWE-ID***	***CWE description*** [89]
ELT	Ether Lost in Transfer [17]	Missing *address* validity checks.	CWE-20 (Class)	*Improper Input Validation*.
RV	Requirement Violation [14]	Improper input validation conditions.		
SA	Short Addresses [13]	Improper validation of the *address* length.		
Atx	Authorization through tx. origin [13]	Improper authorization restriction using *tx origin.*	CWE-284 (Pillar)	*Improper Access Control*.
UEW	Unprotected Ether Withdrawal [14]	Improper access control in Ether withdrawal.		
Usd	Unprotected selfdestruct [14]	Self-destruction with improper authorization checks.		
UWSL	Unprotected Write to Storage Location [14]	Improper authorization checks permit arbitrary writings to storage locations.		
VEF	Visibility of Exposed Functions [13]	Improper access control or authorization permits improper function usage.		
GR	Generating Randomness [17]	Use of predictable random numbers.	CWE-330 (Class)	*Use of Insufficiently Random Values*
MPRA	Missing Protection against Signature Replay Attack [14]	Missing check or protection in data authenticity.	CWE-345 (Class)	*Insufficient Verification of Data Authenticity*
SM	Signature Malleability [14]	Improper verification of data signature.		
Ty	Type Casts [17]	Improper verification of data validity.		
CPL	DoS costly Patterns and Loops [13]	Improper management of resources (*gas*) in pattern and loop execution.	CWE-400 (Class)	*Uncontrolled Resource Consumption*
Gs	Gasless send [17]	Improper check in the usage of gas using *send*.		
BU	Blockhash Usage [17]	Blockhash usage in critical operations exposes to manipulation from miners.	CWE-668 (Class)	*Exposure of Resource to Wrong Sphere*.
ML	Malicious Libraries [90]	Inappropriate access to resources.		
SF	Secrecy Failure [13]	Anyone can accede to a private variable.		
TD	Timestamp Dependency [17]	Timestamp usage in critical operations exposes to manipulation from miners.		
CU	Call to the Unknown [17]	Low-level function calls can be unintended controlled from the resources of another sphere.	CWE-669 (Class)	*Incorrect Resource Transfer Between Spheres*.
DUC	Delegatecall to the Untrusted Callee [13]	Low-level function calls can provide unintended control to a resource of another sphere.		
EC	DoS by External Contracts [90]	External contracts can cause unintended control from a resource of another sphere.		
AP	Arithmetic Precision Order [91]	Divide before multiply can lead to incorrect results.	CWE-682 (Pillar)	*Incorrect Calculation*
IOU	Integer Overflow or Underflow [14]	Overflow or underflow can lead to incorrect results.		
AJ	Arbitrary Jump [14]	Execution of unexpected instructions.	CWE-691 (Pillar)	*Insufficient Control Flow Management*.
FE	Freezing Ether [13]	Modification of the program flow makes it impossible to send Ether.		
IGG	Insufficient gas griefing [14]	Prevention sub-call execution alters the program flow..		
Re	Reentrancy [17]	A callee calls the function back before its completion.		
RLO	Right Left Override [14]	The standard flow of the program is modified.		
TOD	Transaction Ordering Dependence [13]	An attacker can artificially favor the execution of one transaction over another.		
UEB	Unexpected Ether Balance [14]	Strict Ether balance assumptions cause an unexpected program flow.		
ED	Exception Disorder [17]	Incorrect handling of Solidity exception propagation.	CWE-703 (Pillar)	*Improper Check or Handling of Exceptional Conditions*
Us	Unchecked send [13]	Improper checks of exceptional conditions using *send*.		
UV	Unchecked Call Return Values [13]	Missing checks of return values.		

4.1 Characteristics of static analysers

The main characteristics of SA tools are their input format, the internal representation extracted from the code, and the analysis methodologies applied.

4.1.1 The input

SA has the code of the smart contract to be checked. We

- *Bytecode* is a list of compiled instructions executed in an *Ethereum Virtual Machine* (EVM);
- *Source code* refers to the smart contracts high-level programming language (e.g., Solidity).

4.1.2 The internal representation

The internal representation is about the abstract model extracted from the code for the analysis. Alternatives here are the

abstract syntax tree (AST) or *the control flow graph* (CFG):

- AST extracts an abstract representation of the source code by lexical and syntax analysis.
- CFG is a directed graph representing the program flow derived from the AST or the bytecode. The basic blocks of a program serve as nodes. An arc connects node A to node B if block B is executed immediately after block A. The arc labels represent the condition of the path execution.

4.1.3 Methodologies

Methodologies represent the algorithmic approach that tools use to analyse smart contracts for identifying vulnerabilities.

- *Decompilation* (DEC) transforms the bytecode into a language at a higher abstraction level (like an intermediate or Solidity-like language) to enhance the code's readability.
- *Disassembly* (DIS) transforms the EVM bytecode into an assembler language divided into blocks and assigns labels (e.g., to jump destinations and addresses).
- *Symbolic execution* (SE) uses symbols instead of real values of variables, based on determining the path code's reachability through constraints controlled by *Satisfiability Modulo Theories* (SMT) solvers.
- *Taint analysis* (TA) follows the information flow generated from an information source. Initially, only deriving data from the source are considered contaminated. The method keeps track of how this taint propagates (it can happen, e.g., through assignment operation).

4.2 Tools Selection Criteria

At first, an extensive search of different sources as research papers (e.g., [19], [20]) and other online resources (sites for developers of the Ethereum environment and GitHub's most referenced tools repositories) produced a shortlist of 38 candidate tools.

The final set of 9 tools (Table 2) selected for detailed analysis included only those fulfilling the following criteria: 1) handling of contracts written in Solidity version 0.5 or higher; 2) stand-alone tools targeting vulnerabilities detection; 3) analysis of smart contracts without user-defined properties or assertions; 4) free public availability (for a white box analysis discussed later).

A * in the table indicates that specific options extend the support of the original Solidity tool to the 0.5 release.

For the sake of completeness, the list of tools excluded due to the violation of one or multiple selection criteria consists of E-EVM [21], Erays [22], ETHBMC [23], EtherTrust [24], EthIR [25], eThor [26], GasChecker [27], Gasper [28], KEVM [29], MadMax [30], MAIAN [31], Manticore [32], Octopus [33], Porosity [34], Rattle [35], SASC [36], sCompile [37], SIF [38], SmartEmbed [39], SmartInspect [40], SmartBug [41], SolAnalyzer [42], SolGraph [43], SolHint [44], SolMet [45], solc-verify [46], Vandal [47], Verisol [48], Zeus [49].

A short description of the selected tools follows:

- *Securify (Sfy)* uses antipatterns to decide if a software behaviour is safe or unsafe, supported by a domain-specific language [50]. Fallback functions, libraries, and abstract contracts are not supported. We used the main branch release [51].
- *Securify2* (Sfy2) represents a development of Securify taking a Solidity file as input and supporting only flat contracts. The tool decompiles the stack-oriented bytecode into an assignment-based form and transforms the code to DataLog. We used the release of the main branch [52].
- *Slither* (Sli) converts Solidity smart contracts into an intermediate representation called SlithIR [53]. We downloaded the 0.7.0 release from [54].
- *SmartCheck* (SmC) identifies smart contracts vulnerabilities by searching specific source code antipatterns [55]. The tool converts the code into an XML syntax tree, and Xquery path expressions retrieve the vulnerable patterns. We used the master branch [56].
- *Remix-IDE (Rmx)* is continuously under development [56]. Based on different modules, it also performs a static analysis on that we focus. By transforming the code into an AST representation, it checks the presence of unsafe patterns. We installed the release of March 2021.
- *Mythril* (Myt) uses symbolic execution based on EVM bytecode for Ethereum and other EVM-compatible blockchains [57].
- *Oyente* (Oye) is a precursor in the field [58], and several other projects have used it as a starting and reference point. It uses symbolic execution. We used the master branch [59].
- *Osiris* (Osi) extends *Oyente*'s fault model by integer overflows and underflows [60]. It combines symbolic execution and taints analysis. The version downloaded [61] extends Oyente 0.2.7.
- *HoneyBadger* (HoB*)* is another *Oyente*-based tool that employs symbolic execution and a set of heuristics to pinpoint specific vulnerabilities in smart contracts [62]; we used the release based on Oyente 0.2.7 [63].

Table 2
Selected tools.

Tools		Analysis Methods		
Name	**Release**	***Input***	***Internal representation***	***Methodology***
Securify2 [52]	Mai 2020	BC	CFG	DEC, DIS
Securify [50]	Mai 2020	BC	CFG	DEC, DIS
Slither [91]	0.7.0	SC	AST, CFG	TA
SmartCheck [90]	2.1	SC	AST	DEC
Remix IDE [92]	March 2021	SC	AST	Various
Mythril [57]	0.22.17	BC	CFG	SE
Oyente [58]	November 2020	BC	CFG	SE
Osiris [60]	0.0.1*	BC	CFG	SE
HoneyBadger [93]	0.0.1*	BC	CFG	SE

4.3 Preliminary evaluation

The detection capabilities and diagnostic resolution of the individual tools differ significantly. In addition, their diagnostic messages lack a uniform and comparable form. This way, their comparison necessitates mapping the individual targeted sets of weaknesses and diagnostic messages into a consistent basis. This needed a white-box reverse engineering approach. We performed a qualitative analysis that (i) identifies the checking rules applied, (ii) maps them into weaknesses or vulnerabilities, and (iii) identifies the classes of vulnerabilities remaining uncovered.

Checking rules have a slightly different meaning, specified as

follows. Rules of *Securify* represent the checks of the violation of patterns. *Securify2* provides a list of checked antipatterns. Slither's rules designate the different detectors that identify anomalies in the smart contracts. *SmartCheck*'s checking rules are determined by evaluating all Xpaths that the tool can check; the Xptah represents a rule and a specific pattern and can be retrieved as output when it identifies anomalies. *Mythril* and *Remix* rules consider the output of each module of the SA. The number of rules of *Oyente*, Osiris, and *HoneyBadger* represents the number of different outcomes they provide.

We use examples, descriptions, definitions, recommendations, exploits, and other sources linked to checking rules (into the documentation and sometimes into the tool's code) to identify the ones related to the vulnerabilities (the tools also identify other defects). Finally, we map the checking rules to vulnerabilities (and consequently to classes).

The white-box analysis used all the program and documentation-related information sources to reverse engineer the checking rules and their relevance for vulnerability detection.

Table 3 highlights, for each tool, the resulting rules and the vulnerability-related ones, showing how many vulnerabilities the tool can identify (and the number of classes involved).

Table 4 shows instead the vulnerability classes escaping detection completely. An 'X' in cell (z, y) means that the tool in row z did NOT aim to find any vulnerability belonging to the class in column y.

Table 3
Analysis of the internal codes of the tools.

Tools	Resulting rules	Vulnerabilities related rules	Vulnerabilities involved	Classes involved
Sfy	10	10	7	5
Sfy2	43	25	18	7
Sli	71	25	17	8
SmC	85	42	17	8
Rmx	22	13	10	6
Myt	17	15	13	7
Oye	7	7	6	5
Osi	10	7	6	4
HoB	9	5	5	3

Analysing the table, we can observe some facts:

Table 4
Anticipated and uncovered vulnerabilities.

Tools	CWE-20	CWE-284	CWE-330	CWE-345	CWE-400	CWE-668	CWE-669	CWE-682	CWE-691	CWE-703
Sfy			X	X	X	X				
Sfy2			X	X	X					
Sli				X	X					
SmC	X		X							
Rmx	X		X	X				X		
Myt	X			X	X					
Oye	X	X	X	X	X					X
Osi	X	X	X	X	X					X
HoB	X	X	X	X	X	X				X

- *Securify, Securify2,* and *Slither* are the only ones capable of detecting vulnerabilities in CWE-20 (*Improper Input Validation*).
- *CWE-330* (*Use of Insufficiently Random Values*) can be detected only by *Mythril and Slither.*
- Only *SmartCheck* investigates CWE-345 (*Insufficient Verification of Data Authenticity*) and two tools (*SmartCheck* and *Remix*) CWE-400 *(Uncontrolled Resource Consumption).*

Table 4 highlights that no tool in the selection covers by design the entire fault model; thus, diagnostic coverage metrics need a more specific resolution of the individual classes intended to be covered by it. We refer to this subset of the anticipated fault model as the working domain of the particular tool

Besides the view based on classes, we investigated the individual vulnerabilities and found a few that the entire set of SATs cannot detect.

Vulnerabilities out of the detection capabilities of our set of static analysis tools:

Missing Protection against Replay Attack (CWE-345): it occasionally depends on a specific function of the ERC-20 token [64].

Vulnerabilities-related weaknesses related to specific constructs of Solidity that would permit to catch Malicious Libraries (CWE-668), *Requirement Violation* (CWE-20), *Type Casts* (CWE-345), and *Insufficient Gas Griefing* (CWE-691).

5 Dataset and manual inspection

5.1 The reference dataset

We built a reference dataset of smart contracts, extracting randomly smart contracts (around 400) from the Etherscan (the Ethereum blockchain explorer) [65].

A smart contract can contain different logic contracts (contract keyword) that include functions (function keyword). A measuring the size and complexity of logical contracts or functions of smart contracts helps to highlight their essential characteristics.

We define the metrics of the size of the contract as follows: *Physical lines of code* (LOC) is the total number of lines of a smart contract's code. *Logical lines of code* (LLOC) identify the number of lines of code that are neither comments nor empty. These measures indicate the length of a program; the programming style strongly influences them.

Checking a contract under test may result in a successful t*est run* or a *processing failure* - an improper or incomplete test run with partial (or no) diagnostic outcome -. A successful test run may deliver a *no-vulnerability found message* (*negative* result) or a *vulnerability indication* complemented with diagnostic information on the location and type. At first, we focus only on the vulnerability detection capabilities and refer to all cases with a vulnerability indication as a *positive*.

We observed many successful runs and sporadic errors in processing smart contracts using our selected SA tools for the reference dataset. Processing errors happen mainly for two reasons: (i) the tool uses an external module that exhausts memory resources (e.g., *Securify2*); (ii) the tool covers the Solidity language only incompletely.

Table 5
Pilot set characteristics.

Smart Contracts IDs	Characteristics				
	LOC	*LLOC*	*Logic Contracts*	*Functions*	*Vulnerable rows*
0x01a5c0	190	133	1	22	13
0x16eb29	891	298	9	54	22
0x1cdcc3	776	408	8	113	27
0x239669	244	121	4	24	24
0x4b89f8	929	650	8	41	27
0x5571d1	297	145	4	24	10
0x605cc9	640	185	4	37	14
0x607620	180	149	4	14	25
0x999999	488	252	1	18	42
0xaa4de9	709	269	3	37	22
0xbc205b	1267	616	7	85	27
0xbd3149	227	108	5	22	23
0xd82556	1031	561	8	53	45
0xe042c2	1299	521	12	80	42
0xfd77ef	564	268	5	43	48
Total	9732	4684	83	667	411

5.2 The pilot set

The lack of standard benchmarks to compare Solidity checking tools necessitates the creation of a pilot set, a set of representative smart contracts with known vulnerabilities as a ground truth comparison basis. For instance, without additional knowledge, we cannot differentiate between *true positives* (TP – correct detection of an existing vulnerability) and *false positives* (FP – false detection of a non-existing vulnerability) and quantify the vulnerability detection precision of the individual tools.

Thus, we extracted a subset of contracts from the reference dataset for manual inspection, which constitutes our *pilot set* to assemble a *ground truth*[*] [66]. Smart contracts selected from the reference dataset into the pilot set had to fulfil the following constraints inspired by benchmarks principles.

1. *Representativeness of typical domains*: At least one contract of the set must belong to each of the top 5 categories: gambling, exchange, games, finance, properties [67].
2. *Compliance*: All tools can process all contracts without errors.
3. *Representativeness to Solidity*: contracts must contain the main features of the language (e.g., functions that exchange Ether, assembly usage, low-level calls, libraries, structures, arrays).
4. *Representativeness of the vulnerabilities*: The whole set of contracts must contain all kinds of vulnerabilities predicted by SA tools on the reference dataset.

Although the number of contracts in the pilot set is limited to 15, the number of logic contracts is 88 with a total LLOC of 4684 is comparable to the dataset used in other works [41].

The total number of existing vulnerabilities is 486. Moreover, a single line may contain multiple vulnerabilities. Accordingly, we consider vulnerable rows instead of total vulnerabilities. After finding a vulnerability, a manual inspection is required to apply a countermeasure. Thus, the probability of finding other existing ones is very high. The number of vulnerable rows in the pilot set is 411. Table 5**Error! Reference source not found.** summarises the main characteristics of the contracts in the pilot set.

Table 6 provides the total number of vulnerabilities (in the first row) and vulnerable rows (in the second one) per class. In the rest of the paper, the term vulnerability refers to the vulnerable rows.

5.3 The upper limit of class coverage

Table 6
Pilot set vulnerabilities grouped by CWE-IDs.

Analysis	CWE-IDs								
	CWE-20	*CWE-284*	*CWE-330*	*CWE-400*	*CWE-668*	*CWE-669*	*CWE-682*	*CWE-691*	*CWE-703*
Tot. Vuln. (486)	119	256	4	12	24	6	12	34	19
Vul. Rows (411)	74	235	4	12	24	4	11	29	18

The manual inspection identified the vulnerable rows for each class. The classwise ratio between the number of vulnerabilities detectable by a particular tool and the ones present in the pilot set indicates the coverage's upper limit, assuming vulnerabilities are uniformly distributed (Table 7).
Class CWE-345 is omitted from the table because no vulnerability in their reference and pilot set belongs to this class. The following section investigates the quantification of the testing performance

Table 7
The upper limit of class coverage.

Tools	CWE-20	CWE-284	CWE-330	CWE-400	CWE-668	CWE-669	CWE-682	CWE-691	CWE-703
Sfy	1.0	0.2	0.0	0.0	0.0	1.0	0.0	0.9	1.0
Sfy2	1.0	1.0	0.0	0.0	0.6	1.0	0.1	1.0	1.0
Sli	0.6	0.8	1.0	0.0	0.6	1.0	0.1	0.9	1.0
SmC	0.0	0.8	0.0	1.0	1.0	1.0	0.1	0.3	1.0
Rmx	0.0	0.0	0.0	0.9	0.9	1.0	0.0	0.7	0.0
Myt	0.0	0.1	1.0	0.0	0.6	1.0	1.0	0.7	1.0
Oye	0.0	0.0	0.0	0.0	0.6	1.0	1.0	0.8	0.0
Osi	0.0	0.0	0.0	0.0	0.6	1.0	1.0	0.8	0.0
HoB	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.2	0.0

of the individual static analysers.

6 Testing performance

As shown previously in Table 4, no tool in the selection covers by design the entire set of vulnerabilities; thus, diagnostic coverage metrics need a more fine granular resolution of the individual anticipated classes. We refer to this subset of the anticipated anomaly classes as the *working domain* of the particular tool. A detection is missing (for a specific tool) if a vulnerability from its working domain escapes detection (false negative). We use this pilot set, and the purpose of this experimental campaign is to assess how tools perform in their

[*] Data are uploaded in the submission process. In case of acceptance, data will be available online through Zenodo or a similar platform.

respective working domain. In addition, we provide a benchmark evaluating each SA tool as a *black box* (**RQ2**).

6.1 Basic definitions

Vulnerability detection is a classification task. Accordingly, we use the standard statistical metrics for binary classification to quantify the behaviour of the individual tools. The *confusion matrix* is a 2x2 matrix used to describe the classifier behaviour. Columns and rows represent the actual values and classifier results. The matrix's main diagonal represents the correct classification in terms of *true positives* (TP) - correct detection of an existing vulnerability - and *true negatives* (TN) - correct assessment of no vulnerability -. The anti-diagonal matrix contains the false classification: *false negatives* (FN – missed detection of an existing vulnerability) and *false positives* (FP – false detection of a non-existing vulnerability).

Based on the confusion matrix, several performance indicators can be defined [68]. *Recall (R)* is the percentage of detected anomalies (true positives) over all the anomalies (true positives + false negatives):

$$R = TP \div (TP + FN) \quad (1)$$

The term *coverage* is often used with the same meaning of recall. In the rest of the paper, we use the term coverage.

Precision (P) is the ratio of true and total fault indications penalising false alarms:

$$P = TP \,/\, (TP + FP) \quad (2)$$

F_1 *score* is the harmonic mean between precision and recall, defined as follows:

$$F_1 = 2 \times P \times R \,/\, (P + R) \quad (3)$$

Accuracy (A) describes the ratio of correct diagnostic results among all trials:

$$A = (TP + TN) \,/\, (TP + TN + FP + FN) \quad (4)$$

However, accuracy can be misleading when used for unbalanced data, like those we expect for SA tools with dominating true positives. *Balanced accuracy (BA)* [69] normalises true negatives and true positives prediction, as the average of the true positive and negative rates:

$$BA = (TP \,/\, (TP + FN) + TN \div (FP + TN) \,/\, 2 \quad (5)$$

6.2 How tools perform in their working domain

Combining the evaluation of tools in Section 4.3 and the findings described in Section 5.3, we can build confusion matrices for each taxonomy class (not shown for space reason). Table 8 and Table **9** provide coverage and precision for each tool and class, respectively. Light-red cells (x, y) indicate that the tool in row x chose NOT to detect vulnerabilities of the class y. In both tables, light-green cells highlight the best tool x for each class y.

Through the tables, developers can determine how well their tools perform on the vulnerabilities they decided to target. Some observations follow:

- *Slither* has a high precision (>= 0.9) in five classes and has a high coverage (>= 0.9) in two.
- *Securify2 and Slither* perform well regarding precision and coverage in two classes (CWE-284 and CWE-669). *Securify2* covers the class CWE-703 completely (but with low precision) and has the highest coverage in CWE-20.
- *Mythril* has complete coverage and perfect precision in CWE-330.
- *Remix* has good coverage and high precision in CWE-669 and CWE-691.

Next, we want to assess the aggregate coverage and precision of the tools without distinguishing among classes. We generate the

Table 8
Classwise vulnerability coverage – tool working domain.

Tools	CWE-20	CWE-284	CWE-330	CWE-400	CWE-668	CWE-669	CWE-682	CWE-691	CWE-703
Sfy2	0.5	0.9	0.0	0.0	0.0	0.8	0.1	0.2	1.0
Sfy	0.2	0.6	0.0	0.0	0.0	0.0	0.0	0.2	0.4
Sli	0.2	0.9	0.0	0.0	0.5	0.8	0.0	1.0	0.7
SmC	0.0	0.9	0.0	0.4	0.0	0.3	0.0	0.3	0.0
Rmx	0.0	1.0	0.0	0.7	0.7	0.8	0.0	0.8	0.0
Myt	0.0	0.8	1.0	0.0	0.3	1.0	0.0	0.1	0.4
Oye	0.0	0.0	0.0	0.0	0.0	0.5	0.3	0.1	0.0
Osi	0.0	0.0	0.0	0.0	0.0	0.3	0.4	0.1	0.0

Table 9
Precision of tools.

Tools	CWE-20	CWE-284	CWE-330	CWE-400	CWE-668	CWE-669	CWE-682	CWE-691	CWE-703
Sfy2	0.5	0.8	0.0	0.0	0.0	1.0	0.3	0.3	0.2
Sfy	0.6	0.3	0.0	0.0	0.0	0.0	0.0	0.5	0.8
Sli	1.0	1.0	0.0	0.0	0.9	1.0	0.0	0.7	1.0
SmC	0.0	0.9	0.0	1.0	0.0	1.0	0.0	0.3	0.0
Rmx	0.0	0.5	0.0	1.0	0.8	1.0	0.0	0.9	0.0
Myt	0.0	0.8	1.0	0.0	0.8	0.4	0.0	0.5	0.9
Oye	0.0	0.0	0.0	0.0	0.0	1.0	0.6	0.3	0.0
Osi	0.0	0.0	0.0	0.0	0.0	1.0	0.7	1.0	0.0

Fig. 1. Confusion matrices of 6 tools in the tool working domain. Every matrix highlights TP, FP, TN, FN, starting from the bottom left, proceedings clockwise.

Table 10
Tools performances – working domain.

Pilot Set	Sfy2	Sfy	Sli	SmC	Rmx	Myt	Oye	Osi
Precision	0.63	0.42	0.93	0.79	0.86	0.68	0.60	0.50
Coverage	0.72	0.42	0.77	0.72	0.74	0.33	0.11	0.15
Balanced accuracy	0.84	0.70	0.88	0.86	0.87	0.66	0.56	0.57
F1 Score	0.67	0.42	0.84	0.76	0.80	0.44	0.19	0.23

new confusion matrices (Fig. 1) and summarise the main derived metrics in Table 10. Green and light green indicate the top two performances in the tools working domain. *Slither* has the best performances, while *Remix* has the second-best ones.

In this section, we analysed the behaviour of the tools in their working domain. Once we identified the tool's working domains and the maximum coverage in the pilot set, we quantified their classwise and aggregate detection capabilities (Tables Table **8**-Table **10**).

6.3 Tools benchmarking

This section extends our benchmark on the capabilities of the individual tools in the general case evaluating the outcomes obtained by processing the entire pilot set. We use the same four performance metrics as above. This way, because of the definitions in (2), precision is the same as in the previous section (Table 9); the other performance metrics differ due to the relaxation of constraints.

Table 11 provides the coverage for each tool and class. These results can help choose the best tool once specific vulnerabilities are targeted. Some observations follow:

- *Securify2* has the highest coverage in classes CWE-20, CWE-284, CWE-703 (with low precision).
- *Slither* and *Remix* have the highest coverage respectively in classes CWE-669 and CWE-691, CWE-400 and CWE-668 (with high precision).
- *Mythril* and *Osiris* have the highest coverage in the class CWE-330 and CWE-682.

Table 12 provides the overall benchmark results. Green and light-green cells highlight identify the best and second-best results. Main findings follow:

- *No tool* can cover more than 66% of the vulnerabilities in the pilot set (coverage), and no tool exceeds the value of 70% (*Slither*) in the F1 score.

Table 11
Classwise vulnerability coverage – tool benchmarking.

Tools	CWE-20	CWE-284	CWE-330	CWE-400	CWE-668	CWE-669	CWE-682	CWE-691	CWE-703
Sfy2	0.4	0.9	0.0	0.0	0.0	0.7	0.1	0.2	1.0
Sfy	0.2	0.2	0.0	0.0	0.0	0.0	0.0	0.2	0.4
Sli	0.1	0.7	0.0	0.0	0.3	0.7	0.0	0.9	0.7
SmC	0.0	0.7	0.0	0.4	0.0	0.2	0.0	0.1	0.0
Rmx	0.0	0.0	0.0	0.7	0.6	0.7	0.0	0.1	0.1
Myt	0.0	0.1	1.0	0.0	0.2	0.5	0.0	0.1	0.3
Oye	0.0	0.0	0.0	0.0	0.0	0.5	0.3	0.1	0.0
Osi	0.0	0.0	0.0	0.0	0.0	0.2	0.4	0.1	0.0

Table 12
Tools performances – benchmark.

Pilot Set	Sfy2	Sfy	Sli	SmC	Rmx	Myt	Oye	Osi
Precision	0.63	0.42	0.93	0.79	0.86	0.68	0.60	0.50
Coverage	0.66	0.18	0.55	0.44	0.10	0.06	0.01	0.02
Balanced accuracy	0.81	0.58	0.78	0.71	0.55	0.53	0.51	0.51
F1 Score	0.64	0.25	0.70	0.56	0.19	0.11	0.03	0.04

- *Securify2* has the best coverage (even if it misses 1/3 of vulnerable rows) and the second F1 score performance. It has the highest balanced accuracy; however, it has low precision.
- *Slither* has a coverage of 0.10 lower than *Securify2*, but it has a very high precision (0.93). Moreover, it has the second performance (0.78) of balanced accuracy.
- *SmartCheck* has lower performances than Slither by about 0.10 in every metric.
- *Dedicated tools with a specific purpose* and limited working domain have worst performances than general-purpose ones when analysing a broad set of vulnerabilities.

This evidence permits us to argue that considering the coverage alone (disregarding the false positives cost), *Securify2* is a good choice. If we target a balance between precision and coverage, *Slither* is preferred. However, the low diagnostic accuracy (coverage) demands an investigation on the potential improvement using a combination of tools.

7 Coverage improvement

Is it possible to improve the coverage by using a combination of tools? (**RQ3**) As the previous experiments indicated highly different classwise detection capabilities of the individual tools, we may consider combining them.

There are several ways to combine tools [70]. We use multiple SA tools to test a smart contract from the pilot set in our experiment and consider the combined test result *positive* if any of the tools have a positive outcome (OR type combination). This way, fault coverage improves, at a price to increase the number of false positives.

T*rue positives* are rows in which at least one of the tool's predictions is positive, and the row contains at least one vulnerability. *False positives* are rows where at least one of the tool's predictions is positive but contains no vulnerabilities. *False negatives* and *true negatives* follow accordingly.

We run our experiments for all the possible combinations of 2 or 3 tools and report in Table 13 the combinations with the highest fault coverage.

More *False positives* to be analysed (i.e., the precision of the combination is less than that of the best tool) is the cost paid for using the combination. As an asymptotic result, using all the

Table 13
Tools combination.

Pilot set	Sfy2 Sli	Sfy2 SmC	Sfy2 Rmx	Sfy2 Myt	Sfy2 Sli SmC	Sfy2 Sli Rmx
Precision	0.67	0.62	0.65	0.64	0.64	0.68
Coverage	0.81	0.76	0.71	0.68	0.88	0.85
Balanced accuracy	0.89	0.86	0.84	0.82	0.92	0.91
F1 Score	0.73	0.68	0.68	0.66	0.74	0.75

considered tools together detects 382 out of the 411 vulnerable lines are, reaching coverage of 0.93. The best coverage combining two tools remains at 0.81. When combining three tools, the coverage reaches at best the value of 0.88 (*Securify2 – Slither – SmartCheck*). The second-best solution (*Securify2 – Slither – Mythril*) has a value of 0.85 with no meaningful difference in *balanced accuracy* and F_1 *score*. We investigated, though not shown in the table, four-tool combinations. We obtained the best result by adding *Remix* or *Mythril* to the best combination of three tools: the coverage and precision have a value of 0.90 and 0.64.

Another evaluation targeted the coverage for each class rather than considering the aggregate. Proper combinations should include: a tool between *Securify2*, *Securify,* and *Slither* for CWE-20; *Mythril* to cover CWE-330; *SmartCheck* or *Remix* for CWE-400. The best combination contains four tools (*Securify2 – Slither – SmartCheck - Mythril*). This combination is also one of the best four-tool combinations found earlier, which appears thus to be the best from the point of view of aggregate and classwise vulnerability coverage.

Finally, we report the cost in terms of execution run time. The architecture used is a Virtual Machine (Ubuntu64 based, 8GB of memory) that ran in a Linux Server with 24 GB. Slither, Remix, and SmartCheck require less than a minute to analyse the entire pilot set. Securify2, Securify, and Mythril need 30 minutes, 2 hours, and 4 hours respectively.

We run some experiments over big contracts (between 1500 and 6300 LOC). The maximum required time for analysing a big smart contract remains within a minute for Slither, Remix, and SmartCheck. Securify2, Securify, and Mythril require a maximum time of one hour, three hours, and seventeen hours.

8 Vulnerabilities prioritisation

Even using combinations of tools, we have undetected vulnerabilities. We want to investigate whether they are all equally critical or if some can be more dangerous. (**RQ4**).

A prerequisite of hazard analysis is a quantification of the severity of vulnerabilities. This step helps identify the most critical undetected vulnerabilities having a top priority in mitigation.

We investigated several prioritisation methods (e.g., [71], [72], [73]) and chose two ([74], [73]) dealing explicitly with vulnerability prioritisation: i) the typical severity of the *Common Attack Pattern Enumeration and Classification* (CAPEC); ii) the severity determined by the *Common Weakness Scoring System* (CWSS) developed by the CWE.

8.1 CAPEC attack patterns

8.1.1 Overview

CAPEC [72] provides a public catalogue of common attack patterns that an attacker uses to exploit known weaknesses. Each pattern captures the design and execution of an attack, thus providing information about the severity and mitigation of the attack. CAPEC uses four levels of abstraction for attack patterns (ordered by increasing level of detail): *category*, *meta*, *standard*, *detailed.*

Currently,† the CAPEC archive contains 546 attack patterns, divided into two main hierarchical views:

- *Mechanisms of attack,* consisting of 9 categories, represent the fundamental mechanisms used to exploit a vulnerability
- *Domains of attack,* represented by six different categories: *software, hardware, communications, supply chain, social engineering, physical security*.

CAPEC attack patterns capture the exploitation of weaknesses. Each attack pattern contains some information, including a description of the attack, relationships to other attack patterns, a *typical severity* (which we will use for prioritisation).

8.1.2 Mapping CAPEC patterns to vulnerabilities

We consider *vuln* as a type of vulnerability that belongs to our taxonomy. To determine the mapping between vulnerabilities and CAPEC attack patterns, we used the following method:

$\forall\ vuln$

1. Considering the known exploit scenarios:
    a. we determined the *mechanism of the attack*;
    b. we extracted some *keywords*;
2. We identified a set of CAPEC patterns (C_set) by performing a keyword search on the CAPEC list, filtered through the attack mechanism.
3. Then, we selected the maximum severity of attack patterns belonging to *C_set*.

We selected the *maximum severity* attack pattern belonging to *C_Sol*.

8.2 CWSS Scoring

8.2.1 Overview

CWSS [73] provides a way to prioritise weaknesses by proposing a methodology combining three groups of metrics: base finding metric (information extracted from the weakness class), attack surface metric (barriers an attacker must overcome), and environmental metric (characteristics of the environment of the weakness). Each type of metric group is composed of several factors that, combined with appropriate weights, determine the metric's subscore. The combination of subscores determines the final CWSS score [75]. CWSS explicitly supports cases of incomplete information (factors taking *unknown* value) and allows ignoring irrelevant factors in the analysed context (*not applicable* value).

8.2.2 CWSS calculation

For the scoring calculation, we followed [73]. Our basic assumption is that in case of a vulnerability, an attacker can always discover and exploit it. The main factors to consider are:

- *Common consequences* of the CWE-ID (associated with the vulnerability-related weakness) leading to potential technical impacts on the Blockchain;
- *Worst-case scenarios* in terms of business impact;
- *Vulnerability mitigation capability* provided by an internal (e.g., mandatory software construct) or external control (e.g., EVM).

† At the moment of writing, the last CAPEC update is related to 13th January 2022.

The resulting score is a value from 0 to 100, then reported on a scale from 0 to 10.

8.3 CAPEC and CWSS severity

CAPEC uses four severity classes (*critical*, *high*, *medium*, *low*), CWSS returns a score from 0 to 10. We determine the CWSS severity class using the following score-to-severity conversion (used by NIST [71]): 0 - 3.9 *low*; 4.0 - 6.9 *medium*; 7.0 - 8.9 *high*; 9.0 - 10 *critical*. Table 14 shows the vulnerability and classes severity of our taxonomy using the acronyms in Table 1.

The identified severity classes are independent of the set of smart contracts analysed. The maximum severity of the vulnerabilities that belong to the class determines the criticality C (*critical*), H (*high*), M (*medium*) of a class. The critical classes of our taxonomy, identified through both CAPEC and CWSS, are CWE-284, CWE-668, CWE-669, and CWE-691. Moreover, CWSS has two classes that increase the severity from *high* to *critical*: CWE-345 and CWE-682.

Table 14

Severity of vulnerabilities and classes based on CAPEC and CWSS. Severity is indicated with the following colors: blue - critical, red - high, green - medium; grayed out vulnerabilities whose severity is not identified. The first row contains the classes in our taxonomy. The second (CAPEC) and fourth (CWSS) identify the severity of the vulnerabilities. The third (CAPEC) and fifth rows (CWSS) identify the criticality of the class.

CWE-IDs	CWE-20	CWE-284	CWE-330	CWE-345	CWE-400	CWE-668	CWE-669	CWE-682	CWE-691	CWE-703
CAPEC: severity of vuln.	SA RV ELT	Atx UEW Usd VEF UWSL	GR	MPRA SM Ty	CPL Gs	BU TD ML SF	CU DUC EC	IOU AP	RLO FE Re TOD UEB AJ IGG	ED Us UV
CAPEC: class criticality	H	C	M	H	M	C	C	H	C	M
CWSS: severity of vuln.	SA RV ELT	Atx UEW Usd VEF UWSL	GR	MPRA SM Ty	CPL Gs	BU TD ML SF	CU DUC EC	IOU AP	RLO FE Re TOD UEB AJ IGG	ED Us UV
CWSS: class criticality	H	C	M	C	M	C	C	C	C	H

8.4 Prioritisation of false negatives

Once we have determined the severity, we prioritise vulnerabilities escaping detection (FNs). We process the pilot set with the n-tool combinations of Section 7; the number of false-negative types of each combination, grouped by severity, is shown in Fig. 2 (CAPEC) and Fig. 3 (CWSS). As an example, we examine the case of using a 3 (or more) SA tool combination. This

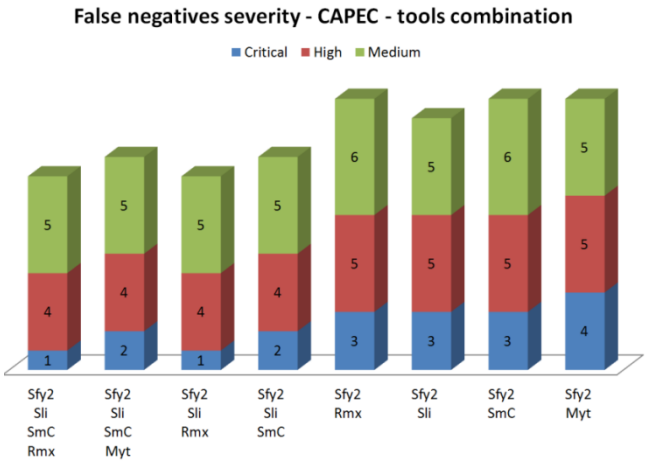

Fig. 2. Types of false negatives - grouped by CAPEC severity - for each combination of tools.

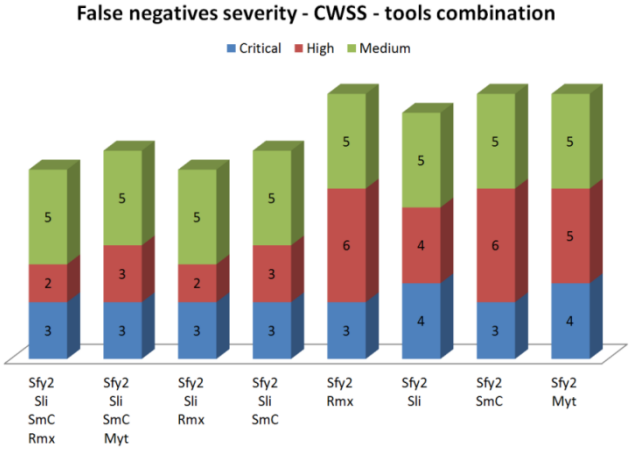

Fig. 3. Types of false negatives - grouped by CWSS severity - for each combination of tools.

setup results in fewer types of false negatives than a 2-tool combination. By considering the union of the FN types in each scoring method, we can argue that:

- CWSS and CAPEC identify 6 FNs with *critical or high* priority (BU, TD, FE, IOU, Re, SA);
- CWSS and CAPEC identify 5 FNs with *medium* priority (CPL, ELT, GR, SF, VEF);
- CWSS and CAPEC identify 0 FNs with *low* priority;
- The two methods differ in identifying FNs with critical and high priority.

In general, technical and business impacts dominate the CWSS score. The values of these factors reflect the particular nature of the Blockchain and the resulting criticality that each breach entails. CAPEC cannot account for this specificity: the severity is a consequence once the pattern is determined. To summarise, CWSS focuses on consequences, CAPEC on the attack method.

By analysing the two figures, we can refine the considerations in *Section 7*:

- the *Sfy2-Sli-SmC-Myt four-tool* combination, although it has fewer occurrences of FNs than *Sfy2-Sli-Rmx*, contains *more* critical or high severity types;
- *The two 4-tool combinations* are equivalent in coverage and

precision; however, including *Remix* contains fewer types of FNs with high or critical severity.

9 Generalisation and treat to the validity

The quality of smart contracts has a significant influence on our analysis results. For instance, widely used quality estimators, like *COnstructive QUALity MOdel* (COQUALMO [76]), use many influencing factors. The sample set of smart contracts considered in our study originated from a public repository with no information on such essential factors as the developers' skills. Qualified people in software security-oriented companies possessing a sophisticated background could produce better contracts regarding software weaknesses and vulnerabilities than the cuff developers. Variance in skills influences both the total number and distribution of the vulnerabilities in smart contracts.

The uncovered classes and vulnerabilities of Table 4 are independent of the smart contract selection. Consider now precision and sensitivity (coverage) in an entire dataset. Then, we highlighted the dependence of the positives in the pilot and reference sets on the contract complexity if the complexity is moderate.

We suppose that eliminating over-represented contracts makes the distribution of CWE classes in a general dataset and pilot set comparable. Tools have different precision and coverage in each class (as highlighted in Table 8 and 9). Thus, precision and coverage in a set of smart contracts depend on the distribution of CWE classes. In our set, smart contracts from the reference dataset, which are processed avoiding over-representing contracts of similar repeated characteristics, form the *reduced* dataset (around 300 smart contracts). The distribution of the positives found by the tools and grouped into CWE classes of a *reduced dataset* is comparable to distributions in the *pilot set,* as shown in Fig. 4. Then, with these premises, findings on precision and coverage are generalisable.

There are several ways to combine tools whereby different choices lead to different results. As shown, using OR in the decision function increases TPs and FPs while using AND decreases TPs and FPs. More complex decision functions could pursue different objectives [70]. The choice made here fits to improve the coverage.

10 Related work

Vulnerabilities and tool analysis are active research areas in different domains, as static analysis is a traditional approach in various software dependability and security areas.

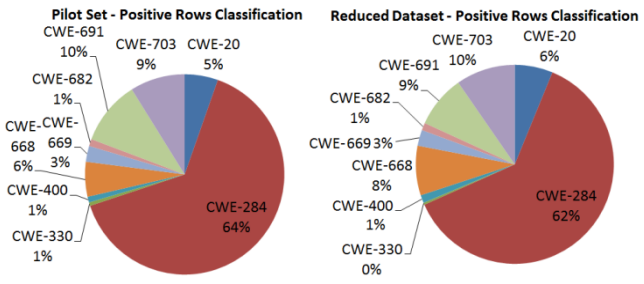

Fig. 4. Distribution of positives in the pilot and reduced dataset).

Atzei et al. [17] presented the first systematic investigation into Ethereum's smart contract vulnerabilities and attacks. It provided a taxonomy of 12 types of vulnerabilities, retrieved from academic literature, analysing attacks and possible countermeasures. Tikhomirov et al. [55] divided 20 kinds of smart contract bugs (we identified 15 vulnerabilities) into four groups (security, functional, operational, and developmental), analysing the severity of several bugs. Pinna et al. [15] performed a comprehensive empirical study of Ethereum smart contracts to provide an overview of smart theirs features, such as type of transactions, the role of the development community, and the source code characteristics. Praitheesan et al. [20] first analysed vulnerability and their detection methods, then focused on the attacks that caused severe losses, finally classified the analysis methods into three categories (static, dynamic, formal analysis). Chen et al. [13] analysed and compared vulnerabilities at the application level, data layer, and network layer. Besides, it investigated the relationship between causes and vulnerabilities and the possible attacks and their impacts for each type of problem explored.

Concerning software security, it is not always possible to analyse all vulnerabilities that have escaped detection; it is crucial to prioritise their analysis. Eschelbeck [77] analysed the problem of prioritising software vulnerabilities within the vulnerability management process. The work provided general observations on the importance of first patching the vulnerabilities with the highest severity. In a first step, Liu et al. [78] compared different prioritisation methods using vulnerabilities extracted from the CVE database. In a second phase, vulnerabilities were grouped by type (CWE-based); the experiments showed an improvement in scoring quality using vulnerability type.

Static analysis is one of the most widely used code checking methods, capable of detecting code vulnerabilities at a relatively low cost. Emanuelsson and Nillson [79] performed one of the first comparisons among static analysers to handle industrial applications. McLean [80] focused on open source tools analysing several programming languages (e.g., C, C++, Java). It performed an analysis of a pre-identified set of vulnerabilities, providing findings and uncovered vulnerabilities. More recently, Nunes et al. [81] proposed a benchmark for web application static analysers based on different criticalities levels, providing a general approach in benchmarking when different scenarios have to be analysed.

The research into applying static analysis to detect Ethereum smart contracts' vulnerabilities increased significantly after the first infamous exploits in 2016 (e.g., DAO [82]), starting from Oyente [58]. Thus, several works compared static analysers. Parizi et al. [83] carried out an experimental assessment of static smart contracts testing tools (Mythril, Oyente, Securify, and Smartcheck) on a set of real-life smart contracts. Durieux et al. [41] analysed 69 contracts considering a set of 10 vulnerabilities based on the Decentralised Application Security Project (DASP) repository [84]. The analysis has been provided through a framework (SmartBug) that parallelises different tools' execution and compares them. Ghaleb et al. [85] implemented SolidFI, a systematic method for automatically evaluating smart contract analysis tools using fault injection. Zhang et al. [86] focused on analysing several bugs grouped in 9 categories based on the IEEE classification for software anomalies. They provided a framework to develop new tools for smart contract analysis. More recently, Dias et al. [87] studied the effectiveness of the three SATs on a set of defects, classified according to the Orthogonal Defect Classification, as defined in [88].

None of the previous studies focused on Solidity >=0.5, how the tools perform in their working domain, the best combination of tools, or prioritisation of FNs. Generally, several works analysed bugs and vulnerabilities (e.g., [85], [86], [87]) without a specific focus on the security of the smart contracts. Some works compared static and dynamic analysis tools (e.g., [86]) when focusing on static analysis capabilities. Moreover, there is a general lack in a deep analysis of tools capabilities, mostly referring to the different classes of vulnerabilities (e.g., [86]). In addition, some works analyzed a small set of vulnerabilities (e.g., [41], [83]), other papers used a small set of tools ([83], [41], [87]) or provided no ground truth ([87], [41]).

Unlike others, our study evaluates smart contracts security, focusing specifically on static analysers (9) and Solidity vulnerabilities (33) classified based on CWE language-independent behaviour. We identified which class of vulnerabilities each tool does not detect and the vulnerabilities uncovered by the whole SATs. Compared with the other works, we investigate coverage and precision (detailed for each model class) and diagnostic accuracy considering what each tool proposes to do and providing a benchmark. The paper suggests a combination of tools for coverage improvement with an analysis of the cost of the combinations and analysing the prioritisation of false negatives.

11 Conclusion

The manuscript focused on the vulnerability-related weaknesses detection capabilities of a collection of static analysers for Solidity. First, the paper provided a fault model for Solidity, classifying a set of Solidity vulnerabilities to a CWE-based taxonomy. Then, it performed a comparative analysis of the selected SA tool capability on a representative set of smart contracts. This analysis allowed us, among others, to quantify the testing efficiency of individual static analysers, improve the coverage using a combination of multiple tools, and prioritise escaping vulnerabilities.

The results show that the contracts used in Ethereum have several vulnerabilities. Considering what they propose to do, only in specific classes do the tools perform well (and thus are well built for those classes). This analysis serves as a guide for developers to increase the impact of tools in smart contract security. Considering tools as black boxes, we quantitatively determined that using a single tool (even the best performing one) is not a good idea if security is the goal. Instead, a combination of tools is preferable (a combination of four tools achieves a coverage of 0.9). Prioritisation allowed us to identify which vulnerabilities that escape the tool combinations should be analysed first due to higher severity. These analyses serve as a guide for users to make smart contracts more secure before deployment.

We have assessed the vulnerability class coverage by the most well-known static analysers. Based on these results, future works can define and apply specific countermeasures against the vulnerabilities which are escaped detections.

REFERENCES

[1] S. Nakamoto, “Bitcoin: a peer-to-peer electronic cash system,” 2008. https://bitcoin.org/bitcoin.pdf.

[2] M. Staderini, E. Schiavone, and A. Bondavalli, “A Requirements-Driven Methodology for the Proper Selection and Configuration of Blockchains,” in *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, 2018, pp. 201–206, doi: 10.1109/SRDS.2018.00031.

[3] N. Szabo, “Formalizing and Securing Relationships on Public Networks,” *First Monday*, vol. 2, n.9, 1997, doi: https://doi.org/10.5210/fm.v2i9.548.

[4] G. Wood, “Ethereum: a secure decentralised generalised transaction ledger,” 2014.

[5] Ethereum, “Solidity Documentation,” *Ethereum foundation*, 2021. https://solidity.readthedocs.io/en/latest/ (accessed Mar. 01, 2021).

[6] ethereum.org, “Ethereum Development Documentation,” 2021. https://ethereum.org/en/developers/docs/ (accessed Nov. 30, 2021).

[7] ISO/IEC, “ISO/IEC 5055:2021 - Information technology — Software measurement — Software quality measurement — Automated source code quality measures.” 2021.

[8] The MITRE Corporation, “CVE.” https://cve.mitre.org/ (accessed Dec. 10, 2021).

[9] National Institute of Standards and Technology, “NATIONAL VULNERABILITY DATABASE.” https://nvd.nist.gov/ (accessed Dec. 10, 2021).

[10] V. Okun, W. F. Guthrie, R. Gaucher, and P. E. Black, “Effect of static analysis tools on software security: Preliminary investigation,” 2007, doi: 10.1145/1314257.1314260.

[11] Center for Assured Software - NSA, “Juliet Test Suite v 1.2 for Java - User Guide,” 2012.

[12] M. Staderini, C. Palli, and A. Bondavalli, “Classification of Ethereum Vulnerabilities and their Propagations,” in *2020 Second International Conference on Blockchain Computing and Applications (BCCA)*, Nov. 2020, pp. 44–51, doi: 10.1109/BCCA50787.2020.9274458.

[13] H. Chen, M. Pendleton, L. Njilla, and S. Xu, “A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses,” *ACM Comput. Surv.*, vol. 53, no. 3, pp. 1–43, Jul. 2020, doi: 10.1145/3391195.

[14] SmartContractSecurity, “SWC Registry,” 2020. https://swcregistry.io/ (accessed Nov. 11, 2021).

[15] A. Pinna, S. Ibba, G. Baralla, R. Tonelli, and M. Marchesi, “A Massive Analysis of Ethereum Smart Contracts Empirical Study and Code Metrics,” *IEEE Access*, vol. 7, pp. 78194–78213, 2019, doi: 10.1109/ACCESS.2019.2921936.

[16] MITRE, “CWE-1000: Research Concepts,” 2021. https://cwe.mitre.org/data/definitions/1000.html (accessed Oct. 27, 2021).

[17] N. Atzei, M. Bartoletti, and T. Cimoli, “A Survey of Attacks on Ethereum Smart Contracts (SoK),” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (2017)*, vol. 10204 LNCS, Springer Verlag, 2017, pp. 164–186.

[18] MITRE, “CWE Mapping & Navigation Guidance,” 2018. https://cwe.mitre.org/documents/cwe_usage/mapping_navigation.html (accessed Jan. 20, 2021).

[19] M. Di Angelo and G. Salzer, “A survey of tools for analyzing ethereum smart contracts,” *Proc. - 2019 IEEE Int. Conf. Decentralized Appl. Infrastructures, DAPPCON 2019*, pp. 69–78, 2019, doi: 10.1109/DAPPCON.2019.00018.

[20] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, “Security analysis methods on ethereum smart contract vulnerabilities - a survey,” 2020. Accessed: Dec. 24, 2020. [Online]. Available: https://arxiv.org/abs/1908.08605.

[21] R. Norvill, B. B. F. Pontiveros, R. State, and A. Cullen, “Visual emulation for Ethereum’s virtual machine,” *IEEE/IFIP Netw. Oper. Manag. Symp. Cogn. Manag. a Cyber World, NOMS 2018*, pp. 1–4, 2018, doi: 10.1109/NOMS.2018.8406332.

[22] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey, “Erays: Reverse engineering Ethereum’s opaque smart contracts,” in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 1371–1385.

[23] J. Frank, C. Aschermann, and T. Holz, “ETHBMC: A bounded model checker for smart contracts,” *Proc. 29th USENIX Secur. Symp.*, pp. 2757–2774, 2020.

[24] I. Grishchenko, M. Maffei, and C. Schneidewind, “EtherTrust: Sound Static Analysis of Ethereum bytecode,” *Tech. Univ. Wien, Tech. Rep*, pp. 1–41, 2018, [Online]. Available: https://www.netidee.at/sites/default/files/2018-07/staticanalysis.pdf.

[25] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, “EthIR: A Framework for High-Level Analysis of Ethereum Bytecode,” *Lect. Notes Comput. Sci. (including Subser. Lect.*

Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 11138 LNCS, pp. 513–520, 2018, doi: 10.1007/978-3-030-01090-4_30.

[26] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, "EThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts," 2020, doi: 10.1145/3372297.3417250.

[27] T. Chen *et al.*, "GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts," *IEEE Trans. Emerg. Top. Comput.*, pp. 1–1, 2020, doi: 10.1109/TETC.2020.2979019.

[28] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 2017, pp. 442–446, doi: 10.1109/SANER.2017.7884650.

[29] E. Hildenbrandt *et al.*, "KEVM: A complete formal semantics of the ethereum virtual machine," *Proc. - IEEE Comput. Secur. Found. Symp.*, vol. 2018-July, pp. 204–217, 2018, doi: 10.1109/CSF.2018.00022.

[30] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: surviving out-of-gas conditions in Ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 1–27, 2018, doi: 10.1145/3276486.

[31] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," *ACM Int. Conf. Proceeding Ser.*, pp. 653–663, 2018, doi: 10.1145/3274694.3274743.

[32] M. Mossberg *et al.*, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," *Proc. - 2019 34th IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2019*, pp. 1186–1189, 2019, doi: 10.1109/ASE.2019.00133.

[33] Patrick Ventuzelo, "Octopus - repository." https://github.com/pventuzelo/octopus (accessed Nov. 10, 2021).

[34] M. Suiche, "Porosity: A decompiler for blockchain-based smart contracts bytecode," 2017.

[35] Crytic, "Rattle - repository." https://github.com/trailofbits/rattle (accessed Nov. 10, 2021).

[36] E. Zhou *et al.*, "Security Assurance for Smart Contract," *2018 9th IFIP Int. Conf. New Technol. Mobil. Secur. NTMS 2018 - Proc.*, vol. 2018-Janua, pp. 1–5, 2018, doi: 10.1109/NTMS.2018.8328743.

[37] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang, "sCompile: Critical Path Identification and Analysis for Smart Contracts," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11852 LNCS, 2019, pp. 286–304.

[38] C. Peng, S. Akca, and A. Rajan, "SIF: A Framework for Solidity Contract Instrumentation and Analysis," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, Dec. 2019, pp. 466–473, doi: 10.1109/APSEC48747.2019.00069.

[39] Z. Gao, V. Jayasundara, L. Jiang, X. Xia, D. Lo, and J. Grundy, "SmartEmbed: A Tool for Clone and Bug Detection in Smart Contracts through Structural Code Embedding," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2019, pp. 394–397, doi: 10.1109/ICSME.2019.00067.

[40] S. Bragagnolo, H. Rocha, M. Denker, and S. Ducasse, "SmartInspect: solidity smart contract inspector," in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, Mar. 2018, vol. 2018-Janua, pp. 9–18, doi: 10.1109/IWBOSE.2018.8327566.

[41] 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*, Jun. 2020, pp. 530–541, doi: 10.1145/3377811.3380364.

[42] S. Akca, A. Rajan, and C. Peng, "SolAnalyser: A Framework for Analysing and Testing Smart Contracts," *Proc. - Asia-Pacific Softw. Eng. Conf. APSEC*, vol. 2019-Decem, pp. 482–489, 2019, doi: 10.1109/APSEC48747.2019.00071.

[43] R. Revere, "Solgraph - repository." https://github.com/raineorshine/solgraph (accessed Nov. 10, 2021).

[44] Protofire, "Solhint - repository." https://github.com/protofire/solhint.

[45] P. Hegedűs, "Towards Analyzing the Complexity Landscape of Solidity Based Ethereum Smart Contracts," *Technologies*, vol. 7, no. 1, p. 6, 2019, doi: 10.3390/technologies7010006.

[46] Á. Hajdu and D. Jovanović, "solc-verify: A Modular Verifier for Solidity Smart Contracts," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12031 LNCS, 2020, pp. 161–179.

[47] L. Brent *et al.*, "Vandal: A scalable security analysis framework for smart contracts," *arXiv Prepr. arXiv1809.03981*, 2018.

[48] Y. Wang *et al.*, *Formal verification of workflow policies for smart contracts in azure blockchain*, vol. 12031 LNCS. Springer International Publishing, 2020.

[49] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing Safety of Smart Contracts.," 2018.

[50] 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*, Oct. 2018, pp. 67–82, doi: 10.1145/3243734.3243780.

[51] SRILAB EthZurich, "Securify - repository." https://github.com/eth-sri/securify (accessed Nov. 10, 2021).

[52] SRI Lab. EthZurich, "Securify v2.0 - repository." https://github.com/eth-sri/securify2 (accessed Nov. 10, 2020).

[53] J. Feist, G. Greico, and A. Groce, "Slither: A static Analysis Framework for Smart Contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in SOftware Egineering for Blockchain*, Mar. 2019, pp. 8–15, doi: 10.1109/SANER.2017.7884650.

[54] Crytic, "Slither 0.7.1 - repository." https://github.com/crytic/slither/tree/0.7.1 (accessed Nov. 10, 2021).

[55] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of ethereum smart contracts," in *2018 ACM/IEEE 1st International Workshop on Emerging Trends in Software Engineering for Blockchain SmartCheck:*, 2018, pp. 9–16, doi: 10.1145/3194113.3194115.

[56] SmartDec, "SmartCheck - repository." https://github.com/smartdec/smartcheck (accessed Nov. 10, 2021).

[57] ConsenSys Diligence, "Mythril 0.22.17 - repository." https://github.com/ConsenSys/mythril/tree/v0.22.17 (accessed Nov. 10, 2021).

[58] L. Luu, D. H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the ACM Conference on Computer and Communications Security*, Oct. 2016, vol. 24-28-Octo, pp. 254–269, doi: 10.1145/2976749.2978309.

[59] Enzyme Finance, "Oyente - repository." https://github.com/enzymefinance/oyente (accessed Nov. 10, 2021).

[60] C. F. Torres, J. Schütte, and R. State, "Osiris," in *Proceedings of the 34th Annual Computer Security Applications Conference*, Dec. 2018, vol. D, no. June, pp. 664–676, doi: 10.1145/3274694.3274737.

[61] C. F. Torres, "Osiris - repository." https://github.com/christoftorres/Osiris (accessed Nov. 10, 2021).

[62] C. F. Torres, M. Steichen, and R. State, "The art of the scam: Demystifying honeypots in ethereum smart contracts," 2019, Accessed: Dec. 24, 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/ferreira.

[63] C. Torres, "HoneyBadger - repository." https://github.com/christoftorres/HoneyBadger (accessed Nov. 10, 2021).

[64] F. Vogelsteller and V. Buterin, "EIP-20: ERC-20 Token Standard," 2015. https://eips.ethereum.org/EIPS/eip-20 (accessed Aug. 10, 2020).

[65] Ethereum, "Etherscan - The Ethereum Blockchain Explorer." https://etherscan.io/ (accessed Nov. 08, 2021).

[66] M. Staderini, A. Pataricza, and A. Bondavalli, "JSS Data." 2022, [Online]. Available: https://drive.google.com/file/d/1WTrOsvi4Btbar1CSmid1bDnAIUFQcF1e/view?usp=sharing.

[67] G. A. Oliva, A. E. Hassan, and Z. M. Jiang, "An exploratory study of smart contracts in the Ethereum blockchain platform," *Empir. Softw. Eng.*, vol. 25, no. 3, pp. 1864–1904, May 2020, doi: 10.1007/s10664-019-09796-5.

[68] S. V. Stehman, "Selecting and interpreting measures of thematic classification accuracy," *Remote Sens. Environ.*, vol. 62, no. 1,

pp. 77–89, 1997, doi: 10.1016/S0034-4257(97)00083-7.

[69] K. H. Brodersen, C. S. Ong, K. E. Stephan, and J. M. Buhmann, "The balanced accuracy and its posterior distribution," *Proc. - Int. Conf. Pattern Recognit.*, pp. 3121–3124, 2010, doi: 10.1109/ICPR.2010.764.

[70] F. Di Giandomenico and L. Strigini, "Adjudicators for diverse-redundant components," in *Proceedings Ninth Symposium on Reliable Distributed Systems*, 1990, pp. 114–123, doi: 10.1109/RELDIS.1990.93957.

[71] NIST, "NVD Vulnerability Severity Ratings," 2020. https://nvd.nist.gov/vuln-metrics/cvss (accessed Nov. 11, 2021).

[72] MITRE Corporation, "Common Attack Pattern Enumeration and Classification," 2021. https://capec.mitre.org/ (accessed Nov. 11, 2021).

[73] MITRE, "CWSS - Scoring CWEs," 2018. https://cwe.mitre.org/cwss/cwss_v1.0.1.html (accessed Aug. 10, 2021).

[74] MITRE, "CAPEC - Summary of Use Cases," 2019. https://capec.mitre.org/about/use_cases.html (accessed Aug. 01, 2021).

[75] MITRE Corporation, "Common Weakness Scoring System," 2018. https://cwe.mitre.org/cwss/cwss_v1.0.1.html (accessed Nov. 11, 2021).

[76] S. Chulani and B. Boehm, "Modeling Software Defect Introduction and Removal: COQUALMO (COnstructive QUALity MOdel)," *USC-CSE Tech. Rep.*, pp. 99–510, 1999.

[77] G. Eschelbeck, "The laws of vulnerabilities: Which security vulnerabilities really matter?," *Inf. Secur. Tech. Rep.*, vol. 10, no. 4, pp. 213–219, 2005, doi: 10.1016/j.istr.2005.09.005.

[78] Q. Liu, Y. Zhang, Y. Kong, and Q. Wu, "Improving VRSS-based vulnerability prioritization using analytic hierarchy process," *J. Syst. Softw.*, vol. 85, no. 8, pp. 1699–1708, 2012, doi: 10.1016/j.jss.2012.03.057.

[79] P. Emanuelsson and U. Nilsson, "A Comparative Study of Industrial Static Analysis Tools," *Electron. Notes Theor. Comput. Sci.*, vol. 217, no. C, pp. 5–21, 2008, doi: 10.1016/j.entcs.2008.06.039.

[80] R. K. McLean, "Comparing static security analysis tools using open source software," *Proc. 2012 IEEE 6th Int. Conf. Softw. Secur. Reliab. Companion, SERE-C 2012*, pp. 68–74, 2012, doi: 10.1109/SERE-C.2012.16.

[81] P. Nunes, I. Medeiros, J. C. Fonseca, N. Neves, M. Correia, and M. Vieira, "Benchmarking Static Analysis Tools for Web Security," *IEEE Trans. Reliab.*, vol. 67, no. 3, pp. 1159–1175, 2018, doi: 10.1109/TR.2018.2839339.

[82] D. Siegel, "Understanding The DAO Attack," 2016. https://www.coindesk.com/learn/2016/06/25/understanding-the-dao-attack/ (accessed Nov. 08, 2021).

[83] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh, "Empirical Vulnerability Analysis of Automated Smart Contracts Security Testing on Blockchains," *Proc. 28th Annu. Int. Conf. Comput. Sci. Softw. Eng.*, pp. 103–113, 2018, doi: 10.5555/3291291.3291303.

[84] NCC Group, "Decentralized Application Security Project (or DASP) Top 10," 2018. https://dasp.co/ (accessed Sep. 10, 2020).

[85] 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*, Jul. 2020, pp. 415–427, doi: 10.1145/3395363.3397385.

[86] P. Zhang, F. Xiao, and X. Luo, "A Framework and DataSet for Bugs in Ethereum Smart Contracts," *Proc. - 2020 IEEE Int. Conf. Softw. Maint. Evol. ICSME 2020*, pp. 139–150, 2020, doi: 10.1109/ICSME46990.2020.00023.

[87] B. Dias, N. Ivaki, and N. Laranjeiro, "An Empirical Evaluation of the Effectiveness of Smart Contract Verification Tools," 2021.

[88] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, B. K. Ray, and D. S. Moebus, "Orthogonal Defect Classification—A Concept for In-Process Measurements," *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 943–956, 1992, doi: 10.1109/32.177364.

[89] Mitre, "CWE Common Weakness Enumeration." https://cwe.mitre.org/ (accessed Oct. 30, 2020).

[90] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain - WETSEB '18*, May 2018, pp. 9–16, doi: 10.1145/3194113.3194115.

[91] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," *Proc. - 2019 IEEE/ACM 2nd Int. Work. Emerg. Trends Softw. Eng. Blockchain, WETSEB 2019*, pp. 8–15, 2019, doi: 10.1109/WETSEB.2019.00008.

[92] Ethereum, "Remix Project - repository." https://github.com/ethereum/remix-project (accessed Nov. 10, 2021).

[93] C. Torres, "HoneyBadger." https://github.com/christoftorres/HoneyBadger (accessed Jun. 02, 2020).