Evaluating The Vulnerability Detection Efficacy Of Smart Contracts Analysis Tools

Silvia Bonomi[1][0000−0001−9928−5357], Stefano Cappai[1], and Emilio Coppa[2][0000−0002−8094−871X]

[1] Sapienza University of Rome, Rome, Italy
bonomi@diag.uniroma1.it, cappai.1844363@studenti.uniroma1.it
[2] LUISS University, Rome, Italy. ecoppa@luiss.it

Abstract. Smart contracts on modern blockchains pave the way to the development of novel application design paradigms, such as Distributed Applications (DApps). Interestingly, even some safety-critical systems are starting to adopt such a technology to devise new functionalities. However, being software, smart contracts are susceptible to flaws, posing a risk to the security of their users and thus making crucial the development of automatic tools able to spot such flaws.
In this paper, we examine 11 real-world DApps that participated in security auditing contests on the Code4rena platform. We first conduct a manual analysis of the vulnerabilities reported during the contests and then assess whether state-of-the-art analysis tools can identify them. Our findings suggest that current tools are unable to reason on business logic flaws. Additionally, for other root causes, the detectors in these tools may be ineffective in some cases due to a lack of generality or accuracy. Overall, there is a significant gap between auditors' findings and the results provided by these tools.

Keywords: Smart Contract · Vulnerability · Testing Tools · Blockchain.

1 Introduction

Distributed Ledger Technologies (DLTs) and Decentralized Applications (DApps) [12, 10], implemented through smart contracts, have become essential components in various modern solutions across diverse application domains, ranging from finance to agriculture. Among all the interested domains from this technological advent, there are also safety-critical systems [24] that are starting to adopt blockchains and smart contracts to support new functionalities. Secure information sharing of sensible data with integrity and confidentiality requirements [13], decentralized access control with privacy guarantee [1], continuous monitoring enabling certified removal of data in safety-critical databases [15] and (food) tracking along the supply chain [34] are just few examples of the increasing presence of DLTs and DApps in domains characterized by really high-quality dependability and security standards. For instance, the community is exploring the use of blockchains in the context of railway industry [35, 22, 32, 25, 26].

In safety-critical systems, more than in other domains, there is a huge attention to the system development life-cycle that should follow *security-by-design* and *continuous*

monitoring principles to ensure that bugs and anomalies are timely detected and resolved. Indeed, the system development is usually supported by guidelines provided by certification standards and whose aim is to give recommendations to developers regarding all the development process activities with a particular emphasis on the *verification and validation* tasks and the maintenance aspects. Thus, it is a crucial requirement for a safety-critical system that its software components are deeply tested to discover the highest number of bugs and vulnerabilities *before* the system starts to be operational and that monitoring is in place when the system gets operational to ensure fast detection and patching at runtime.

This requirement becomes even more relevant if the system includes a software module developed by using smart contracts. Indeed, differently from traditional software, a smart contract is a piece of code deployed on top of a distributed ledger through the execution of a transaction. The direct consequence is that, once it is deployed, it is very hard to patch due to the immutability property of the underlying ledger. The direct consequence is that the adoption of DLTs- and DApps-based solutions in safety-critical systems requires increased attention to the verification and validation phase which translates into the need for testing tools able to detect and discover bugs and vulnerabilities with the highest accuracy possible[3].

In response to these problems, over the past decade, both the research community and the industry have initiated efforts to develop and explore software testing methods and security tools. The goal is to potentially identify bugs and vulnerabilities in smart contracts before deploying these software components on blockchains. Similar to traditional software, various techniques [17, 40, 29, 31, 30, 42, 33, 20, 3, 41] can help developers detect the flaws.

The first aid to developers can come from *lightweight static analysis* tools [17, 40], which can provide quick feedback about common security issues by performing a local pattern-driven analysis. Being quite efficient, the community is starting to integrate them even in continuous integration setups. However, they can be inaccurate, missing crucial security flaws and reporting a large number of invalid reports, i.e., false positives. To mitigate such problems, the community has explored different *heavyweight analysis* frameworks based, e.g., on symbolic execution [29, 31, 30] and fuzzing [42, 33, 20], aiming at better accuracy in exchange for worse scalability. Finally, formal approaches [3, 41] have been investigated to bring stronger security guarantees but often require rethinking the blockchain technologies or imposing constraints on the development.

As for traditional software, the community has launched several security auditing platforms, such as Code4rena and Immunefi, where DApp developers can ask experts to assess the quality and security of smart contracts in exchange for monetary rewards.

Considering the history of DLTs and smart contracts and the actual spreading of these technologies between different application domains, most of the auditing are performed on applications developed for the financial domain. However, many of the existing vulnerabilities are not context-dependent (i.e., they affect the Solidity programming language or the distributed environment hosting the blockchain) and thus the capability of a testing tool to discover it has a global interest beyond the financial domain (e.g.,

[3] https://ethereum.org/en/developers/docs/smart-contracts/formal-verification/

Access Control Vulnerabilities in Solidity Smart Contracts may lead to sensitive data exposure in the healthcare domain).

Our contributions. In this practical experience paper[4], we examine 11 real-world DApps that participated in security auditing contests on the Code4Arena platform from March to August 2023. We first conduct a manual analysis of the vulnerabilities reported during the contests by the auditors to understand what are the most common security issues emerging in real-world smart contracts. These contracts are particularly interesting because, being part of well-known and relevant DApps, have been implemented by experienced developers. We then report the results of running three state-of-the-art tools on these DApps, assessing how their findings fare against auditors' discoveries. In particular, we considered two lightweight static analysis tools, namely *SmartCheck* and *Slither*, and one heavyweight analysis framework based on symbolic execution called *Mythril*. Our assessment suggests that current tools are unable to reason on business logic flaws, the most frequent root cause of vulnerabilities for the 11 DApps that we considered. Additionally, for other root causes, the detectors in these tools are ineffective in some cases due to a lack of generality or accuracy. Overall, unfortunately, there is still a significant gap between auditors' findings and the results provided by these tools suggesting the need for new developments in this context.

2 Background

Distributed Ledger Technologies and Blockchains. DLTs are an emerging class of decentralized distributed systems that allow the recording of transactions of assets in a digital ledger. Among them, blockchains gained huge popularity and are currently the most widespread DLT. A blockchain, as the name suggests, is a particular type of ledger where transactions are collected and stored in blocks and blocks are linked among them using pointers i.e., inserting in a block i the hash of the content of previous block $i-1$. Each block is constructed by selecting submitted transactions and validating their correctness and consistency with respect to the current ledger state. Once a block is created, it needs to be chained to the current ledger. This is done collaboratively by participants in the distributed systems that need to agree on the *next* block becoming the head of the chain (i.e., on the last ledger state). Blockchains are currently attracting a lot of attention due to their *immutability* property i.e., once a new block is created and attached to the chain, it can not be easily altered. This is achieved by combining and using together cryptographic primitives and robust consensus algorithms.

The Ethereum blockchain. Currently, many different blockchain implementations exist but currently the most widespread can still be considered the Ethereum blockchain [39]. Firstly proposed by Vitalik Buterin in late 2013 the network became live in 2015 with its native cryptocurrency called *Ether* (ETH). Since then, it evolved adapting its internal mechanism and supporting additional features like smart contracts. The Ethereum cryptocurrency is used to incentivise participants who perform computations and validate transactions and it can also be used to pay for transaction fees and services on the Ethereum network. In Ethereum, every operation on the network requires

[4] This paper is an extended version of a preliminary 2-page fast abstract presented at ISSRE 2023 [8] where the analysis was restricted to only 4 DApps and a single tool i.e., Mythril.

a certain amount of computational resources that are measured in *gas units* i.e., the gas is the unit to measure the amount of computational effort required to execute operations or run smart contracts. Users must pay for gas in Ether when they execute transactions.

Smart contracts. A smart contract is a self-executing contract (i.e., a piece of running software) with the terms of the agreement directly written into code. It is executed on top of a blockchain and it automatically runs its functions when predefined conditions are met by generating transactions on the blockchain. Smart contracts are written in programming languages specifically designed for blockchain platforms, such as *Solidity* for Ethereum. Smart contracts are deployed over all the blockchain nodes and are executed by all the participants. This decentralization ensures that the contract's execution is transparent, secure, and resistant to censorship or manipulation.

Ethereum supports the execution of smart contracts thanks to a Turing-complete virtual machine called the *Ethereum Virtual Machine* (EVM). Once deployed, smart contracts operate autonomously and independently of any human intervention. In addition, they are *immutable* i.e., once the code is deployed on the blockchain, it cannot be easily altered, patched or tampered with. While on one hand, such immutability ensures the integrity and reliability of the contract's execution, on the other hand, it raises potential significant dependability and security issues due to bugs and vulnerabilities.

3 Related Works

Smart contracts are currently used across many different domains ranging from DeFi where they are used to offer financial services[5] to *Supply Chain Management* where smart contracts are used to improve transparency and traceability of products and goods along the supply chain. This fast and large spreading is also pushing the scientific community to investigate security aspects connected to DApps. In particular, several smart contract analysis tools have been proposed by the community building on top of different techniques, such as *lightweight static analysis* [17, 40, 29], *symbolic execution* [31, 30], *fuzzing* [42, 33, 20], and other *verification* techniques [3, 41].

Different experimental studies have already tried to assess and compare the detection capabilities of such tools. In [43], authors analyzed several Code4rena contests using Oyente [29], a symbolic execution framework which is, however, now deprecated and unmaintained. They split findings into two groups: *Machine Auditable Bugs (MABs)* and *Machine Unauditable Bugs (MUBs)*. Their study aims to show the current state of automatic tools, and how those tools seem not able to find MUB vulnerabilities. Our contribution continues this investigation by enlarging the set of analyzed contests and taking an orthogonal perspective, trying to shed light on the strengths and weaknesses of different automatic tools when compared with human auditors. In [4], authors analyze thousands of smart contracts already deployed on the blockchain, i.e., they scan the bytecode of those contracts. They propose the *skeleton* concept: they found common patterns in the bytecode of smart contracts, grouping them accordingly. However, this approach could flatten the dataset excessively since most smart contracts are characterized by external calls to others, which often could be the actual source of vulnerabilities.

[5] Examples of DeFi DApps are Compound, Aave, Uniswap, and MakerDAO.

Moreover, by just looking at the bytecode, it could be very hard to identify some types of vulnerabilities. In our study, we consider also the Solidity code of the DApps, including tools that can process both Solidity and the resulting bytecode.

4 Experimental Study Methodology

This section presents key aspects characterizing our experimental study: the source for our dataset (i.e., the auditing platform and the considered contests), the analysis tools, and our experimental setup.

Security Auditing Platform. *Code4rena (C4)* is a community-driven competition for smart contract audits. Any DApp developer can use it to launch a new contest involving a specific project and define what code is in scope and what is not. There are three main roles on this platform:

- *Wardens:* auditors in charge of reporting issues affecting a DApp. They are typically human experts who, exploiting both manual and automatic tools, produce a detailed report. Wardens are not forced to disclose their analysis strategies;
- *Sponsors:* DApp developers sponsoring the contest with a *prize pool*. The higher the prize pool is, the higher the interest from advanced security auditors will be;
- *Judges:* experts rating the performance of wardens and deciding the severity and validity of their findings. Judges are chosen by the C4 community among the best wardens, according to their impartiality and accountability.

The prize pool is split into the following categories: *High* findings, *Medium* findings, *Quality Assurance (QA)* findings (composed of *Low* and *Non-Critical* findings), *Gas Optimization* findings, *Analysis*, and *Bot Races*. Usually, a *contest* lasts 1-3 weeks. High, Medium, and Quality Assurance findings are reported by wardens within the contest's period. Wardens report these issues establishing the severity. After the submission phase, the contest ends and judges analyze reports of wardens, checking for validity of issues and correctness in severity assignment. *Gas Optimization* findings are suggestions proposed by wardens to rewrite a better code that saves gas. *Analysis* reports are reports that analyze the DApp as a whole, identifying the architecture weaknesses. *Bot Races* are competitions among automatic tools developed by wardens. In the first hour of every contest, bot race participants can submit reports obtained by the execution of their bot. Only a few participants, who have already qualified during monthly bot qualifications, can report findings during the bot races.

Considered Contests. C4 contests can naturally provide a significant dataset for smart contracts. Indeed, they provide complex and real-world contracts that have reached a strong maturity. Moreover, the evaluation performed by the judges over the reported issues can naturally provide a *ground truth* that we can exploit in our experimental study. Overall, we analyzed 11 contests from March to August 2023. Table 1 summarizes the main characteristics of such contests, which involved 73 contracts, 15,562 SLOCs, and 130 vulnerabilities. Our selection of contests was aimed at exposing different *types* of DApps and different complexities (in terms of number of smart contracts and SLOCs). In the remainder of this section, we provide more details about these contests.

DApp	Date	Type	Contracts	SLOCs	Humans		Bots	
					High	Medium	High	Medium
Ajna	2023/05	Lending	3	1,391	11	14	0	2
Asymmetry	2023/03	Derivative	4	460	8	12	0	0
Caviar	2023/04	DEX	4	741	3	17	0	0
Eigenlayer	2023/04	Derivative	7	1,393	2	2	0	0
ENS	2023/04	Service	9	2,022	0	7	0	0
Frankencoin	2023/04	Stablecoin	8	949	6	15	0	0
Juicebox	2023/05	Service	1	160	0	3	0	0
Livepeer	2023/08	Social	4	1,605	2	3	0	0
Llama	2023/06	Service	11	2,047	2	3	0	1
Shell	2023/08	Service	1	460	1	0	0	0
Stader	2023/06	Derivative	21	4,334	1	14	0	1
Overall			73	15,562	36	90	0	4

Table 1: Smart Contracts Dataset Overview.

Ajna. The Ajna protocol [2] is a lending and borrowing protocol. There were 3 main contracts in the contest: `GrantFund`, which holds the treasury, i.e., an amount of tokens that are used for the governance; `PositionManager`, a position of a lender in a given pool; `RewardsManager`, which rewards a lender who decides to stake token.

Asymmetry. Asymmetry [5] aims at providing a solution to the centralization of the staked Ether market through *Liquid Staked Ethereum Derivatives*. The contest involved four smart contracts: `SafEth`, which allow a user to stake some funds; `Reth`, `WstEth`, and `SfrxEth`, that contain methods to acquire rETH, wstETH, and sfrxETH tokens.

Caviar. Caviar [11] is an on-chain, gas-efficient automated market maker (AMM) protocol for trading non-fungible tokens (NFTs), that allow users to deposit NFTs and associated assets inside *Liquidity Pools* (LP). The contest covered three implementation bits of the `Custom Pools`: `Factory`, which creates NFTs and holds protocol fees; `PrivatePool`, which allows a developer to set which operations (buy, sell, exchange) can be performed on the NTFs; `EthRouter`, used to perform actions across pools.

Eigenlayer. Eigenlayer [14] is stacking platform that allows users to (re-)stake their ETHs and ERC20 tokens based on custom strategies, which can then be accepted by validators. The contest involved 7 contracts: `StrategyManager`, which tracks stakers' deposits of tokens; `StrategyBase`, which represents a base strategy; `EigenPodManager`, which allows users to stake ETH; `EigenPod`, which allows users to stake ETHs on Ethereum and restake them on EigenLayer; `DelayedWithdrawalRouter`, which controls withdrawals of ETHs from `EigenPod`s; `Pausable` and `PauserRegistry`, which can extend other contracts and makes them pausable, i.e., stops their tasks.

ENS. The Ethereum Name Service (ENS) [16] is a distributed domain naming system (DNS) based on the Ethereum blockchain. The most interesting contract proposed in the

contest are: `DNSRegistrar`, which is in charge of the domain registration process exploiting a `registry` using a DNSSEC Oracle; `DNSClaimChecker`, which verifies DNS name claims; `OffchainDNSResolver`, which handles the domain name resolution.

Frankencoin. Frankencoin [19] is a collateralized stablecoin token, dubbed ZCHF, that tracks the value of the Swiss Franc. Out of 8 contracts included in the contest, four were extremely crucial: `Position`, which represents the position in tokens for a user; `MintingHub`, which creates positions; `Frankencoin`, which is the actual token; and `StablecoinBridge`, which allows users to swap Frankencoin with other stablecoins.

Juicebox. The Juicebox protocol [23] is a *programmable* treasury, allowing users to automatically mint NTFs when new funds are received in the context of a specific treasury. A single contract (`JBXBuybackDelegate`) was investigated in the contest: it is in charge of distributing tokens to a contributor according to its donation to the treasury.

Livepeer. Livepeer [27] aims at providing a distributed video live-streaming service built on top of Ethereum. The contest covered the handling of the protocol rewards for *broadcasters*, *transconders*, and *orchestrators*: `BondingManager`, which manages the protocol staking and rewards; `Treasury`, which holds funds of treasury and executes proposals; `LivepeerGovernor`, an OpenZeppelin Governor implementation; `BondingVotes`, which is tied to the transcoders selection process.

Llama. Llama [28] provides a governance framework to make life easier for DApp developers. The contest included the main contracts behind the Llama architecture: `LlamaCore`, which checks the execution of *actions*; `LlamaExeutor`, the actual executor of the actions; `LlamaPolicy`, an ERC721 contract that defines roles and permissions.

Shell Protocol. Shell [37] offers a platform, called Ocean, that can compose any type of DeFi primitive: AMMs, lending pools, algorithmic stablecoins, and NFT markets. The contest included only the contract `EvolvingProteus`, which can offer a primitive of the AMM implementation, i.e., a liquidity pool and its evolution over time.

Stader Labs. Stader [38] is a non-custodial staking platform for multiple Proof-of-Stake networks through the liquid staking token ETHx. The contest involved: `ETHx`, the actual ERC20 token; `PermissionedPool` and `PermissionlessPool` handle the deposit of ETHs; `StaderOracle` provides a source of exchange rates; `StaderStakePoolsManager` allows users to stake ETHs, mint ETHxs, manage staking rewards.

Analysis Tools. As discussed in Section 2, there exists a large number of smart contract analysis tools. However, in this experimental study, we decided to focus on tools that are well-known in the Ethereum ecosystem and that have been used in different academic works as well as in security evaluations publicly disclosed by auditors. In particular, we considered one lightweight pattern-based static analysis tool, called *SmartCheck* [40], a more advanced yet still lightweight static analysis tool, called *Slither* [17], and then one advanced heavyweight analysis tool based on symbolic execution [7], called *Mythril* [31]. We did not consider dynamic tools, e.g., fuzzers, or tools using formal verificationdue to their non-trivial setup or lack of support for complex smart contracts.

SmartCheck. SmartCheck is a lightweight static analyzer that translates the Solidity code of a smart contract into an XML-based intermediate representation. Then, it checks the result with XPath patterns to detect four categories of issues: *security*, i.e., issues that may lead to vulnerable states; *functional* and *operational*, i.e., issues affecting the logic

or leading to performance degradation; *developmental*, i.e., issues causing problems at deployment time. Overall, it has 43 issue detectors. We used SmartCheck 0.2.0.

Slither. Slither is Python framework for analyzing smart contracts. By translating solidity code into a custom intermediate representation, called *SlithIR*, it allows to implement detectors. It supports more than 80 detectors, covering different vulnerabilities categories. Moreover, it is quite efficient at performing its analysis and thus developers are starting to consider it within continuous integration setups. We used Slither 0.9.4.

Mythril. Mythril is a symbolic execution framework for EVM bytecode. It explores the program state space of a smart contract by executing its code on symbolic inputs, i.e., inputs whose value is not *apriori* fixed. In particular, an interpreter evaluates the EVM bytecode, constructing formulas to represent the data flows of the program computation. Whenever the program meets a decision point, the framework evaluates using an SMT solver whether the formula related to the decision is feasible, i.e., the smart contract can take that specific path for an assignment of the inputs. Moreover, the framework can use the solver to evaluate conditions related to well-known vulnerable patterns. Mythril integrates 13 vulnerability detectors. In our experiments, we used Mythril 0.23.24.

Smart Contract Vulnerabilities. In the literature, smart contract vulnerabilities have been classified according to different criteria [6, 21, 43, 36] and, in this paper, we group them in four categories [6, 43]:

- **Business Logic**: any issue due to the erroneous application of the *business logic* model. These flaws are strongly tied to the nature of a DApp;
- **Solidity**: any flaw due to the erroneous use of the Solidity language or unexpected nuances of its design, e.g., rounding arithmetic rules;
- **EVM**: issues due to the EVM behavior, e.g. the gas block limit;
- **Blockchain**: problems resulting from the nature of the blockchain, e.g. untrustworthy oracles or dependence on transaction order.

Experimental Setup. We now describe the setup adopted during our study.

We executed our experiments on a server equipped with two Intel Xeon E5-4610v2 CPUs and 256 GB of RAM, running on Ubuntu 22.04. The tools were executed under Docker: for SmartCheck, we exploited the container image from the SmartBugs framework [18], while, for Slither, we installed it on a vanilla Python 3 container image, and finally, for Mythril, we used its official container image.

While SmartCheck and Slither are expected to terminate their analysis in just a few minutes, Mythril, as most symbolic execution frameworks, can easily run for several hours or even days. Hence, we decided to run the first two tools for up to two hours, while leaving Mythril running up to 7 days. Tools were executed in independent experiments and did not share their budget. SmartCheck and Slither do not have critical parameters that need tuning. On the other hand, Mythril can work with five *exploration strategies* (*DFS*, *BFS*, *naive-random*, *weighted-random*, and *pending*), which may affect the generation of program states, possibly bringing different results. Hence, we performed different experiments for each strategy and then considered the best result. Furthermore, Mythril defines a *Max Depth* parameter, which limits the maximum depth for an execution path, thus bounding the analysis over a path, avoiding to waste the entire time budget over a single path. We kept this parameter at its default value but then increased it when investigating the results related to some specific contests.

		Ajna	Asymmetry	Caviar	EigenLayer	ENS	Frankencoin	Juicebox	Livepeer	Llama	Shell	Stader	
Blockchain	Frontrunning	◐ ◐	◧ ◐	◐			◧ ◧ ◐ ◐ ◐			◐		● ◐	13
	Erroneous Assumption						◐						1
Business Logic	Erroneous Logic Model	◐	◧ ◐ ◐	◐	◐		◧ ◧ ◐ ◐ ◐ ◐					◐ ◐ ◐	15
	Freezing Active Position	◧ ◧ ◧	◐				◧ ◐	◐					7
	Erroneous Accounting	◧ ◐ ◐	◧ ◧ ◐	◐ ◐ ◐	◧	◐ ◐ ◐	◐ ◐ ◐	◐	◐	◧		◐ ◐ ◐	22
	Missing Logic Checks	◧ ◧ ◧ ◧ ◐ ◐ ◐	◧ ◐ ◐	◐	◧	◐	◐		◧	◧	◧	◐ ◐ ◐	20
	Wrong Implementation	◐		◐		◐ ◐		◐	◐ ◐	◐		◐ ◐	10
EVM	Gas Limit DoS	■								◐			2
Solidity	Reentrancy			◧ ◐ ◐ ◐									4
	Precision Loss	● ●	◧ ●				◐						5
	Integer Overflow	◧ ◐	◧	◐ ◐			◧		◧				7
	Arbitrarily Code Injection	◧		◧ ◐								■	4
	Outdated Compiler Bug	●											1
	Unsafe Casting	◐		◧		◐							3
	Use of `transferFrom()`	◐											1
	Use of `_mint()`	◐								◐			2
	Dangerous `DelegateCall`												0
	Relying on External Source		■ ● ●	◐ ◐ ◐	●		◐					◐	9
	Division by Zero DoS		◐										1
	Temporal DoS		◐										1
	Erroneous ERC721 Impl.			◐									1
	Erroneous `Pausable` Impl.											◐	1
Overall		27	20	20	4	7	21	3	5	6	1	16	130

Table 2: Overview of the security issues identified by humans, bot, and analysis tools, on the 11 contests, dividing them based on their vulnerability nature. Legend:
□: high severity issue. ◌: medium severity issue.
◧◐: the issue was found only by humans. ◧◐: the issue was found only by bots.
◨◑: the issue was found only by analysis tools. ■●: the issue was found both by humans and tools. ■●: the issue was found both by bots and tools.

5 Experimental Study Results

We now report the results of our experimental study involving the 11 Code4rena contests presented in Section 4. To evaluate the effectiveness of the three state-of-the-art analysis tools, we relied on the reports submitted by humans, i.e., the wardens, and bots to Code4rena, which have been validated by expert judges. Hence, these reports constitute our *ground truth* about the critical issues affecting the smart contracts under auditing. Our investigation has been targeted around the following research questions:

RQ1: Do the contests experience similar security issues?
RQ2: Are the analysis tools considered in this study effective at detecting such issues?
RQ3: Do the results of the tools align with their claimed detection capabilities?
RQ4: Is a more complex analysis able to bring significantly better results compared to a lighter analysis?

5.1 RQ1: Contests versus Vulnerabilities

We start by analyzing the distribution of security issues across the contests. Table 2 provides a visual summary of the security issues that were reported either by a human or by a bot (notice that an issue reported by a bot cannot be reported also by a human). The shape of the symbols in the table represents the severity of the flaw: square for high severity and circle for medium severity. Underlined symbols represent issues originally identified by bots, while not underlined symbols represent problems originally reported by humans. To help perceive the nature of vulnerabilities, we manually classified them according to the groups presented in Section 4.

From a quick look at the table, from top to bottom, we can see that most vulnerabilities, i.e., 74 out of 130, are related to the business logic of a DApp. Then, the second most frequent group, with 40 out of 130 flaws, is the one involving Solidity-specific aspects. Finally, 14 and 2 security problems derive from blockchain and EVM nuances, respectively. When instead we look at the table from left to right, we can observe that different contests were affected by different numbers and types of vulnerabilities. Nonetheless, we can quickly notice that the vast majority of flaws have been found by humans, with only 4 issues reported by bots. In particular, bots mostly identified uses of unsafe functions and their findings were always evaluated as medium-severity issues.

An interesting question is whether the DApp type is somehow connected to the number of discovered flaws. Table 1 reports the type of each DApp. Lending DApps, DEXs, and Stablecoin projects have reported significantly more vulnerabilities than Derivative, Service, and Social DApps. In particular, the first set of DApps has seen more than 19 issues every 1k of SLOC, while the second set has seen less than 7 issues every 1k of SLOC. We hypothesize that the first group contains DApps for which the community has seen several hacks in the past and thus is quite careful with some operations and more trained to recognize some vulnerable patterns. Strangely, Derivative DApps should be part of the first group but instead fall into the second one. Differently, Social DApps are a new thing in the blockchain environment, while Service DApps have a unique behavior and are strongly dependent on the context. Hence, it could be harder for auditors to spot uncommon issues in these types of DApps. Nonetheless, these considerations must be taken lightly due to our limited dataset size.

		Ajna	Asymmetry	Caviar	EigenLayer	ENS	Frankencoin	Juicebox	Livepeer	Llama	Shell	Stader	
Contest Results	Humans	25	20	20	4	7	21	3	5	5	1	15	126
	Bots	2								1		1	4
Tools Results	SmartCheck	■	◐									◧	3
	Slither	● ● ●	■ ● ● ◐		●							◧	9
	Mythril											◧ ●	2

Table 3: Comparison between humans and bots findings versus tools findings.
Legend: □: high severity issue. ◌: medium severity issue.
◧◐: the issue found at least by two tools. ■●: the issue found only by one tool.

Due to the lack of space, we cannot present in detail the vulnerabilities affecting the contests. Hence, we refer to our technical report [9] for a more in-depth discussion.

5.2 RQ2: Tools versus Vulnerabilities

We now include in the scope of our discussion the three analysis tools that we considered in our study. Table 2 visually depicts which security issues, originally reported either by a bot or by a human, have been also detected by at least one of the tools during our experiments: shapes with an empty right side have not been detected by any tool, while full black shapes have been detected also by at least one of the tools under consideration[6].

With a quick look, we can see that tools identified mostly flaws related to Solidity aspects. This makes sense as this group of vulnerabilities is quite well-known and has been targeted for a long time by state-of-the-art smart contracts analysis frameworks. Tools were able to spot a few issues tied to nuances of the blockchain or the EVM. Differently, they were unable to find any security flaw related to the business logic group, which, however, was also the one with more reported vulnerabilities.

A more clear view of the tool efficacy is given by Table 3, where we report the count of findings for each tool. Tools discovered 14 security flaws, among which 11 are unique (see full black shapes in Table 2). In particular, Slither has found most of these flaws (9 out of 14), followed by SmartCheck (3 out of 14) and Mythril (2 out of 14).

An important observation is that, since we are considering publicly available tools, DApp developers may have already run such tools on their projects. Hence, we believe that these tools may have identified additional flaws but they were fixed before the contest. We attempted to recover such information by looking for tool-specific configuration files in the DApp repositories. Interestingly, 7 out of 11 DApps show evidence of Slither, only one DApp was likely tested with Mythril, and none appear to have exploited SmartCheck. However, 5 DApps used Solhint, a common alternative to SmartCheck.

[6] Since we based our ground truth on the validated findings from bots and humans, there is no case where a security issue has been identified only by a tool. Nonetheless, we manually investigated any additional issue reported only by a tool: we could not demonstrate the validity of such (unknown) flaws, i.e., we believe that these findings could be classified as false positives.

Overall, unfortunately, the efficacy of the tools appears to be quite limited compared to the auditors' findings. The next research questions attempt to investigate these results.

5.3 RQ3: Tools in Theory versus Tools in Practice

Given the poor efficacy of the tools at finding the security issues, we investigated whether such tools were equipped with adequate *vulnerability detectors* that could allow them to report the problems that emerged in the contests. SmartCheck integrates 43 detectors, Slither has 83 detectors, and Mythril includes 13 detectors.

Based on our analysis [9], among the 22 vulnerability types listed in Table 2, around 12 of them are targeted by at least one tool detector. Tools mostly consider issues due to Solidity, EVM, and Blockchain, but struggle at devising detectors targeting the business logic[7]. Unfortunately, even when detectors are available for a vulnerability type, they were not always able to correctly find the related flaws for our contests.

One possible reason behind detector failures could be due to our time and memory budget limits. However, both SmartCheck and Slither did not experience any timeout or out-of-memory error. Differently, Mythril experienced several timeouts in the Ajna contest, which may explain some detection failures. For instance, we believe Mythril should be able to find the *Integer Overflow* in Ajna given a sufficient time budget. Estimating such budget is quite hard since this tool explores the program state space, which highly depends on the smart contract complexity: we observed a good correlation between the analysis time and the number of SLOC[8], requiring on average 42 hours for each contest. Given our already large time budget (one week), increasing this limit could be not a practical workaround for most developers. Another reason behind the failures in Mythril could be the impact of the *Max Depth* parameter, which aims at limiting state explosion. To investigate such hypothesis, we repeated the experiments setting the parameter to 1000, which generated additional timeouts without improving the results.

Investigating why the detectors may fail, after ruling out scalability issues, is not trivial as it requires a manual and issue-specific investigation. Due to the lack of space, we only report two case studies. The first one is related to *precision loss*, with five known instances but only three discovered by Slither and SmartCheck, which ship with a detector for this vulnerability type. One missed issue is in Asymmetry, where precision loss arises due to rounding rules in the case of a division after multiplication. However, the tools only consider the inverse pattern, i.e., division before multiplication. Similarly, the tools fail to detect an issue in Frankencoin because the division before multiplication is split across different portions of code, making static analysis harder. Mythril, thanks to its state space analysis, given the proper detector, should identify these flaws. Another interesting case study is related to *integer overflow*, with 7 known instances but zero detections from Mythril, which ships with a detector for this vulnerability type. In Ajna, Mythril fails to detect an overflow because its detector does not cope with OpenZeppelin's SafeMath library, which can handle overflows at execution time but does not avoid them. In particular, when SafeMath is used, Mythril incorrectly skips the overflow analysis.

[7] The only exception is related to *Missing Logic Checks* where Mythril exploits assertions embedded in the code to check specific logic conditions.

[8] Table 1 does not report the SLOC of the imported libraries, which, however, are evaluated by Mythril. For instance, Ajna requires to reason on more than 15,000 SLOC.

Overall, tools may have the potential to improve their detection capabilities, even when considering only non-business logic issues. Mythril has great potential due to its powerful state space analysis but suffers from the lack of detectors and scalability issues, while the other tools may benefit from more general pattern definitions.

5.4 RQ4: Analysis Complexity versus Tool Efficacy

One result emerging from our experiments is that SmartCheck and Slither found more security flaws than Mythril. Moreover, Mythril has not discovered any high-severity flaw. This is unexpected because Mythril is a state-of-the-art symbolic execution framework that may in principle outperform simpler analysis frameworks, such as SmartCheck and Slither, by exchanging scalability for efficacy. However, as already pointed out in RQ3, Mythril does not ship with the same broad set of detectors as the other tools. Nonetheless, looking at the detected flaws is only one side of the story when considering automatic analyses. Indeed, a prominent concern is often the false positive rate. In our experiments, Mythril produced 2 true positives (TP) and 61 false positives (FP), thus with a precision of 3.28%. Slither performed significantly worse, with 9 TPs and 1061 FPs, thus with a precision of 0.46%. SmartCheck positioned between the two, with a precision of 1.46%, 3 TPs, and 202 FPs. Overall, FPs are still a big problem for these three tools, likely hurting their adoption. The more in-depth analysis from Mythril has likely contributed to limiting the number of FPs but there is still room for improvement. For instance, SmartCheck reported more than 30 *Gas Limit DoS* invalid alerts, which can be ruled out when exploiting Mythril analysis. Additionally, we believe that Mythril analysis can likely support the implementation of more powerful detectors, integrating more general patterns. For instance, Mythril discovered a *frontrunning* vulnerability, whose detection is likely out of reach for the two other tools. Similarly, the *precision loss* issues split across two portions of code discussed in RQ3 is likely a good example where a more global analysis can favor the implementation of more robust detectors.

6 Conclusions

In this paper, we report our practical experience when running three state-of-the-art tools on smart contracts from 11 Code4rena contests. We have manually analyzed all the security issues reported during the contests by humans and bots, classifying them in common vulnerability types. Then, we observed that tools are unable to find business logic security flaws, which were the most common vulnerable cause. Our observation is consistent with another recent study [43]. We believe that a key idea to mitigate such limitation would be to ask developers to formalize logic conditions through, e.g., assertions or invariant checkers, allowing tools such as Mythril to exploit them during the analysis. Tools performed better on flaws related to Solidity, EVM, and the blockchain, but their vulnerability detectors come with several limitations in terms of accuracy and generality, which we believe could be mitigated by exploiting powerful analyses such as the one used by Mythril. However, Mythril suffers from scalability issues due to its underlying analysis, suggesting that it is not only a matter of engineering effort. It is interesting to note that, among all analyzed DApps, some vulnerable contracts implement common functions used as building blocks for safety-critical systems (e.g., the tracking

coin logic implemented in Frankecoin is not different from the logic implemented to track contaminated food) and thus the same performance of such tools extends to other domains. There seems to exist still a big gap between the expectations of security levels required from safety-critical systems and the capability of smart contract testing tools.

Acknowledgements

This papers is partially supported by the Sapienza research project DYNASTY (protocol number RM123188F791C0F7).

References

1. Abid, A., Cheikhrouhou, S., Kallel, S., Tari, Z., Jmaiel, M.: A Smart Contract-Based Access Control Framework For Smart Healthcare Systems. The Computer Journal **67**(2), 407–422 (12 2022). https://doi.org/10.1093/comjnl/bxac183
2. Ajna: Paper, https://www.ajna.finance/pdf/Ajna_Protocol_Whitepaper_10-19-2023.pdf
3. Almakhour, M., Sliman, L., Samhat, A.E., Mellouk, A.: Verification of smart contracts: A survey. Pervasive and Mobile Computing . https://doi.org/j.pmcj.2020.101227
4. di Angelo, M., Durieux, T., Ferreira, J.F., Salzer, G.: Evolution of automated weakness detection in ethereum bytecode: a comprehensive study. Empirical Software Engineering (2023). https://doi.org/10.1007/s10664-023-10414-8
5. Asymmetry: Whitepaper (2023), https://www.asymmetry.finance/whitepaper
6. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). In: Principles of Security and Trust (2017). https://doi.org/10.1007/978-3-662-54455-6_8
7. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A Survey of Symbolic Execution Techniques. ACM Computer Surveys (2018). https://doi.org/10.1145/3182657
8. Bonomi, S., Cappai, S., Coppa, E.: On the efficacy of smart contract analysis tools. In: 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE Computer Society (2023). https://doi.org/10.1109/ISSREW60843.2023.00041
9. Bonomi, S., Cappai, S., Coppa, E.: Extended version. Tech. rep. (01 2024), https://github.com/niser93/SmartContractToolAnalysis/blob/master/TechReport.pdf
10. Casino, F., Dasaklis, T.K., Patsakis, C.: A systematic literature review of blockchain-based applications: Current status, classification and open issues. Telematics and Informatics **36**, 55–81 (2019). https://doi.org/https://doi.org/10.1016/j.tele.2018.11.006
11. Caviar: Caviar docs (2023), https://docs.caviar.sh/
12. Chowdhury, M.J.M., Ferdous, M.S., Biswas, K., Chowdhury, N., Kayes, A.S.M., Alazab, M., Watters, P.: A comparative analysis of distributed ledger technology platforms. IEEE Access **7**, 167930–167943 (2019). https://doi.org/10.1109/ACCESS.2019.2953729
13. Díaz, M., Soler, E., Llopis, L., Trillo, J.: Integrating blockchain in safety-critical systems: An application to the nuclear industry. IEEE Access pp. 190605–190619 (2020). https://doi.org/10.1109/ACCESS.2020.3032322
14. EigenLayer: Whitepaper (2023), https://docs.eigenlayer.xyz/assets/files/EigenLayer_-WhitePaper-88c47923ca0319870c611decd6e562ad.pdf
15. Elia, N., Barchi, F., Parisi, E., Pompianu, L., Carta, S., Bartolini, A., Acquaviva, A.: Smart contracts for certified and sustainable safety-critical continuous monitoring applications. In: Chiusano, S., Cerquitelli, T., Wrembel, R. (eds.) Advances in Databases and Information Systems. pp. 377–391. Springer International Publishing, Cham (2022)
16. ENS: Documentation (2021), https://docs.ens.domains/
17. Feist, J., Greico, G., Groce, A.: Slither: a static analysis framework for smart contracts. WETSEB '19 (2019). https://doi.org/10.1109/WETSEB.2019.00008

18. Ferreira, J.a.F., Cruz, P., Durieux, T., Abreu, R.: Smartbugs: a framework to analyze solidity smart contracts. ASE '20 (2021). https://doi.org/10.1145/3324884.3415298
19. Frankencoin: Documentation (2023), https://docs.frankencoin.com/
20. Grieco, G., Song, W., Cygan, A., Feist, J., Groce, A.: Echidna: effective, usable, and fast fuzzing for smart contracts. ISSTA 2020 (2020). https://doi.org/10.1145/3395363.3404366
21. Gupta, B.C., Kumar, N., Handa, A., Shukla, S.K.: An insecurity study of ethereum smart contracts. In: Security, Privacy, and Applied Cryptography Engineering (2020)
22. Hua, G., Zhu, L., Wu, J., Shen, C., Zhou, L., Lin, Q.: Blockchain-based federated learning for intelligent control in heavy haul railway. IEEE Access **8**, 176830–176839 (2020). https://doi.org/10.1109/ACCESS.2020.3021253
23. Juicebox: Documentation (2023), https://docs.juicebox.money/dev/
24. Knight, J.C.: Safety critical systems: challenges and directions. In: Proceedings of the 24th Int. Conf. on Software Engineering. ICSE '02 (2002). https://doi.org/10.1145/581339.581406
25. Kuperberg, M., Kindler, D., Jeschke, S.: Are smart contracts and blockchains suitable for decentralized railway control? Ledger **5** (2020). https://doi.org/10.5195/LEDGER.2020.158
26. Liang, H., Zhang, Y., Xiong, H.: A blockchain-based model sharing and calculation method for urban rail intelligent driving systems. In: 2020 IEEE 23rd Int. Conf. on Intelligent Transportation Systems (ITSC). pp. 1–5 (2020). https://doi.org/10.1109/ITSC45102.2020.9294263
27. Livepeer: Whitepaper (2017), https://github.com/livepeer/wiki/blob/master/WHITEPAPER.md
28. Llama: Documentation (2023), https://docs.llama.xyz/
29. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. CCS '16 (2016). https://doi.org/10.1145/2976749.2978309
30. Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., Dinaburg, A.: Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. ASE 2019 (2019). https://doi.org/10.1109/ASE.2019.00133
31. Mueller, B.: Smashing ethereum smart contracts for fun and real profit (2018), https://github.com/muellerberndt/smashing-smart-contracts/blob/master/smashing-smart-contracts-1of1.pdf
32. Naser, F.: Review: The potential use of blockchain technology in railway applications. In: 2018 IEEE International Conference on Big Data (Big Data) (2018)
33. Nguyen, T.D., Pham, L.H., Sun, J., Lin, Y., Minh, Q.T.: sfuzz: an efficient adaptive fuzzer for solidity smart contracts. ICSE '20 (2020). https://doi.org/10.1145/3377811.3380334
34. Oriekhoe, O.I., Ilugbusi, B.S., Adisa, O.: Ensuring global food safety: Integrating blockchain technology into food supply chains. Engineering Science & Technology Journal **5**(3), 811–820 (Mar 2024). https://doi.org/10.51594/estj.v5i3.905
35. Preece, J., Easton, J.: A Review of Prospective Applications of Blockchain Technology in the Railway Industry. Tech. rep. (2018). https://doi.org/10.13140/RG.2.2.15751.75681
36. Ruggiero, C., Mazzini, P., Coppa, E., Lenti, S., Bonomi, S.: Sok: A unified data model for smart contract vulnerability taxonomies. In: Proceedings of the 19th International Conference on Availability, Reliability and Security. ARES '24 (2024)
37. Shell: The ocean (2022), https://github.com/Shell-Protocol/Shell-Protocol/blob/main/
38. Stader: Documentation (2023), https://www.staderlabs.com/docs-v1/intro
39. defillama team: Total value locked all chains, https://defillama.com/chains
40. Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y.: Smartcheck: static analysis of ethereum smart contracts. WETSEB 2018 (2018)
41. Tolmach, P., Li, Y., Lin, S.W., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification. ACM Comput. Surv. (2021). https://doi.org/10.1145/3464421
42. Torres, C.F., Iannillo, A.K., Gervais, A., State, R.: Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. EuroS&P 2021 (2021)
43. Zhang, Z., Zhang, B., Xu, W., Lin, Z.: Demystifying exploitable bugs in smart contracts. ICSE 2023 (2023). https://doi.org/10.1109/ICSE48619.2023.00061