Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts

Thomas Durieux
INESC-ID and IST, University of Lisbon, Portugal
thomas@durieux.me

João F. Ferreira
INESC-ID and IST, University of Lisbon, Portugal
joao@joaoff.com

Rui Abreu
INESC-ID and IST, University of Lisbon, Portugal
rui@computer.org

Pedro Cruz
INESC-ID and IST, University of Lisbon, Portugal
pedrocrvz@gmail.com

ABSTRACT

Over the last few years, there has been substantial research on automated analysis, testing, and debugging of Ethereum smart contracts. However, it is not trivial to compare and reproduce that research. To address this, we present an empirical evaluation of 9 state-of-the-art automated analysis tools using two new datasets: i) a dataset of 69 annotated vulnerable smart contracts that can be used to evaluate the precision of analysis tools; and ii) a dataset with all the smart contracts in the Ethereum Blockchain that have Solidity source code available on Etherscan (a total of 47,518 contracts). The datasets are part of SmartBugs, a new extendable execution framework that we created to facilitate the integration and comparison between multiple analysis tools and the analysis of Ethereum smart contracts. We used SmartBugs to execute the 9 automated analysis tools on the two datasets. In total, we ran 428,337 analyses that took approximately 564 days and 3 hours, being the largest experimental setup to date both in the number of tools and in execution time. We found that only 42% of the vulnerabilities from our annotated dataset are detected by all the tools, with the tool *Mythril* having the higher accuracy (27%). When considering the largest dataset, we observed that 97% of contracts are tagged as vulnerable, thus suggesting a considerable number of false positives. Indeed, only a small number of vulnerabilities (and of only two categories) were detected simultaneously by four or more tools.

CCS CONCEPTS

• **Software and its engineering** → **Software defect analysis**; **Software testing and debugging**.

KEYWORDS

Smart contracts, Solidity, Ethereum, Blockchain, Tools, Debugging, Testing, Reproducible Bugs

1 INTRODUCTION

Blockchain technology has been receiving considerable attention from industry and academia, for it promises to disrupt the digital online world by enabling a democratic, open, and scalable digital economy based on decentralized distributed consensus without the intervention of third-party trusted authorities. Among the currently available blockchain-based platforms, Ethereum [5] is one of the most popular, mainly because it enables developers to write distributed applications (Dapps) based on smart contracts — programs that are executed across a decentralised network of nodes. The main language used to develop Ethereum smart contracts is Solidity[1], a high-level language that follows a JavaScript-like, object-oriented paradigm. Contracts written in Solidity are compiled to bytecode that can be executed on the Ethereum Virtual Machine (EVM).

Smart contracts are at the core of Ethereum's value. However, as noted by some researchers [3, 27], due to the idiosyncrasies of the EVM, writing secure smart contracts is far from trivial. In a preliminary study performed on nearly one million Ethereum smart contracts, using one analysis framework for verifying correctness, *34,200* of them were flagged as vulnerable [32]. Also, Luu *et al.* [27] proposed the symbolic execution tool Oyente and showed that of *19,366* Ethereum smart contracts analyzed, *8,833* (around 46%) were flagged as vulnerable. Famous attacks, such as TheDAO exploit [11] and the Parity wallet bug [37] illustrate this problem and have led to huge financial losses.

There has been some effort from the research community to develop automated analysis tools that locate and eliminate vulnerabilities in smart contracts [18, 27, 39, 42]. However, it is not easy to compare and reproduce that research: even though several of the tools are publicly available, the datasets used are not. If a developer of a new tool wants to compare the new tool with existing work, the current approach is to contact the authors of alternative tools and hope that they give access to their datasets (as done in, e.g., [35]).

The aim of this paper is twofold. First, to be able to execute and compare automated analysis tools, hence setting the ground for fair comparisons, we provide two datasets of Solidity smart contracts. The first dataset contains 69 manually annotated smart contracts that can be used to evaluate the precision of analysis tools. The second dataset contains all available smart contracts in the Ethereum Blockchain that have Solidity source code available on Etherscan (a total of 47,518 contracts) at the time of writing. We have executed 9 state-of-the-art automated analysis tools on the two datasets and analyzed the results in order to provide a fair point of comparison for future smart contract analysis tools. In total, the execution of all the tools required 564 days and 3 hours to complete 428,337 analyses.

Second, to simplify research on automated analysis techniques for smart contracts, we provide a novel, extendable, and easy-to-use execution framework, called SmartBugs, to execute these tools on the same execution environment. This framework currently contains 9 configured smart contract analysis tools.

[1] Interested readers on Solidity, refer to https://solidity.readthedocs.io.

In summary, the contributions of the paper are: (1) A dataset of annotated vulnerable Solidity smart contracts; (2) A dataset that contains all the available smart contracts from the Ethereum blockchain that have Solidity source code available in Etherscan; (3) An execution framework that includes 9 pre-configured smart contract analysis tools; and (4) An analysis of the execution of 9 tools on 47,587 smart contracts.

Our study demonstrates that there are several open challenges that need to be addressed by future work to improve the quality of existing tools and techniques. We report that the current state-of-the-art is not able to detect vulnerabilities from two categories of DASP10: *Bad Randomness* and *Short Addresses*. Also, the tools are only able to detect together 42% of the vulnerabilities from our dataset of annotated vulnerable smart contracts (48 out of 115). The most accurate tool, *Mythril*, is able to detect only 27% of the vulnerabilities. When considering the largest dataset, 97% of contracts are tagged as vulnerable, thus suggesting a considerable number of false positives. In conclusion, we show that state-of-the-art techniques are far from being perfect, still likely producing too many false positives. On the positive side, the best performing techniques do so at a marginal execution cost.

SmartBugs is available at
https://smartbugs.github.io

2 STUDY DESIGN

Blockchain technologies are getting more and more attention from the research community and also, more importantly, from industry. As more blockchain-based solutions emerge, there is a higher reliance on the quality of the smart contracts. The industry and the research community came up with automatic approaches that analyze smart contracts to identify vulnerabilities and bad practices. The main goal of this paper is to report the current state of the art of currently available automated analysis tools for smart contracts. To facilitate reproducibility and comparison between tools, the study is performed using a new extendable execution framework that we call SmartBugs (see Section 2.4).

In this section, we present the design of our study, including the research questions, the systematic selection of the tools and datasets of smart contracts, the execution framework, and the data collection and analysis methodology.

2.1 Research Questions

In this study, we aim to answer the following research questions:

RQ1. [Effectiveness] What is the effectiveness of current analysis tools in detecting vulnerabilities in Solidity smart contracts?
In this first research question, we are interested in determining how precise state-of-the-art analysis tools are in detecting vulnerabilities in known faulty smart contracts.

RQ2. [Production] How many vulnerabilities are present in the Ethereum blockchain?
In this research question, we investigate the vulnerabilities that are detected in contracts pulled from the Ethereum blockchain. We consider the most popular vulnerabilities, the evolution of the vulnerabilities over time, and the consensus among different combinations of automated analysis tools.

Table 1: Tools identified as potential candidates for this study.

#	Tools	Tool URLs
1	contractLarva [2]	https://github.com/gordonpace/contractLarva
2	E-EVM [33]	https://github.com/pisocrob/E-EVM
3	Echidna	https://github.com/crytic/echidna
4	Erays [44]	https://github.com/teamnsrg/erays
5	Ether [26]	N/A
6	Ethersplay	https://github.com/crytic/ethersplay
7	EtherTrust [19]	https://www.netidee.at/ethertrust
8	EthIR [1]	https://github.com/costa-group/EthIR
9	FSolidM [28]	https://github.com/anmavrid/smart-contracts
10	Gasper [9]	N/A
11	HoneyBadger [41]	https://github.com/christoftorres/HoneyBadger
12	KEVM [21]	https://github.com/kframework/evm-semantics
13	MadMax [17]	https://github.com/nevillegrech/MadMax
14	Maian [32]	https://github.com/MAIAN-tool/MAIAN
15	Manticore [30]	https://github.com/trailofbits/manticore/
16	Mythril [31]	https://github.com/ConsenSys/mythril-classic
17	Octopus	https://github.com/quoscient/octopus
18	Osiris [40]	https://github.com/christoftorres/Osiris
19	Oyente [27]	https://github.com/melonproject/oyente
20	Porosity [38]	https://github.com/comaeio/porosity
21	rattle	https://github.com/crytic/rattle
22	ReGuard [25]	N/A
23	Remix	https://github.com/ethereum/remix
24	SASC [43]	N/A
25	sCompile [6]	N/A
26	Securify [42]	https://github.com/eth-sri/securify
27	Slither [16]	https://github.com/crytic/slither
28	Smartcheck [39]	https://github.com/smartdec/smartcheck
29	Solgraph	https://github.com/raineorshine/solgraph
30	Solhint	https://github.com/protofire/solhint
31	SolMet [20]	https://github.com/chicxurug/SolMet-Solidity-parser
32	teEther [23]	https://github.com/nescio007/teether
33	Vandal [4]	https://github.com/usyd-blockchain/vandal
34	VeriSol [24]	https://github.com/microsoft/verisol
35	Zeus [22]	N/A

RQ3. [Performance] How long do the tools require to analyze the smart contracts?
And finally, we compare the performance of the analysis tools. The goal is to identify which tool is the most efficient.

2.2 Subject Tools

In order to discover smart contract automated analysis tools, we started off by using the survey of Angelo *et al.* [12] and we extended their list of tools by searching the academic literature and the internet for other tools. We ended up with the 35 tools that are listed in Table 1.

Not all the identified tools are well suited for our study. Only the tools that met the following three inclusion criteria were included in our study:

Table 2: Excluded and included analysis tools based on our inclusion criteria.

	Inclusion criteria	Tools that violate criteria
Excluded (26)	Available and CLI (C1)	Ether, Gasper, ReGuard, Remix, SASC, sCompile, teEther, Zeus
	Compatible Input (C2)	MadMax, Vandal
	Only Source (C3)	Echidna, VeriSol
	Vulnerability Finding (C4)	contractLarva, E-EVM, Erays, Ethersplay, EtherTrust, EthIR, FSolidM, KEVM, Octopus, Porosity, rattle, Solgraph, SolMet, Solhint
Included (9)	HoneyBadger, Maian, Manticore, Mythril, Osiris, Oyente, Securify, Slither, Smartcheck	

- *Criterion #1.* [Available and CLI] The tool is publicly available and supports a command-line interface (CLI). The CLI facilitates the scalability of the analyses.
- *Criterion #2.* [Compatible Input] The tool takes as input a Solidity contract. This excludes tools that only consider EVM bytecode.
- *Criterion #3.* [Only Source] The tool requires only the source code of the contract to be able to run the analysis. This excludes tools that require a test suite or contracts annotated with assertions.
- *Criterion #4.* [Vulnerability Finding] The tool identifies vulnerabilities or bad practices in contracts. This excludes tools that are described as analysis tools, but only construct artifacts such as control flow graphs.

After inspecting all 35 analysis tools presented in Table 1, we found 9 tools that meet the inclusion criteria outlined. Table 2 presents the excluded and included tools, and for the excluded ones, it also shows which criteria they did not meet.

HoneyBadger [41] is developed by a group of researchers at the University of Luxembourg and is an Oyente-based (see below) tool that employs symbolic execution and a set of heuristics to pinpoint honeypots in smart contracts. Honeypots are smart contracts that *appear* to have an obvious flaw in their design, which allows an arbitrary user to drain Ether[2] from the contract, given that the user transfers a priori a certain amount of Ether to the contract. When HoneyBadger detects that a contract *appears* to be vulnerable, it means that the developer of the contract wanted to make the contract look vulnerable, but is not vulnerable.

Maian [32], developed jointly by researchers from the National University of Singapore and University College London, is also based on the Oyente tool. Maian looks for contracts that can be self-destructed or drained of Ether from arbitrary addresses, or that accept Ether but do not have a payout functionality. A dynamic analysis in a private blockchain is then used to reduce the number of false positives.

Manticore [30], developed by TrailOfBits, also uses symbolic execution to find execution paths in EVM bytecode that lead to reentrancy vulnerabilities and reachable self-destruct operations.

Mythril [31], developed by ConsenSys, relies on concolic analysis, taint analysis and control flow checking of the EVM bytecode to prune the search space and to look for values that allow exploiting vulnerabilities in the smart contract.

Osiris [40], developed by a group of researchers at the University of Luxembourg, extends Oyente to detect integer bugs in smart contracts.

Oyente [27], developed by Melonport AG, is one of the first smart contract analysis tools. It is also used as a basis for several other approaches like Maian and Osiris. Oyente uses symbolic execution on EVM bytecode to identify vulnerabilities.

Securify [42], developed by ICE Center at ETH Zurich, statically analyzes EVM bytecode to infer relevant and precise semantic information about the contract using the Souffle Datalog solver. It then checks compliance and violation patterns that capture sufficient conditions for proving if a property holds or not.

Slither [16], developed by TrailOfBits, is a static analysis framework that converts Solidity smart contracts into an intermediate representation called SlithIR and applies known program analysis techniques such as dataflow and taint tracking to extract and refine information.

Smartcheck [39], developed by SmartDec, is a static analysis tool that looks for vulnerability patterns and bad coding practices. It runs lexical and syntactical analysis on Solidity source code.

2.3 Datasets of Smart Contracts

For this study, we crafted two datasets of Solidity smart contracts with distinct purposes. The first dataset, SB$^{\text{CURATED}}$, consists of 69 vulnerable smart contracts (see Section 2.3.1). Contracts in this dataset are either real contracts that have been identified as vulnerable or have been purposely created to illustrate a vulnerability. The goal of this dataset is to have a set of known vulnerable contracts labelled with the location and category of the vulnerabilities. This dataset can be used to evaluate the effectiveness of smart contract analysis tools in identifying vulnerabilities.

The second dataset is named SB$^{\text{WILD}}$ (see Section 2.3.2) and contains 47,518 contracts extracted from the Ethereum blockchain. The set of vulnerabilities of those contracts is unknown; however, this dataset can be used to identify real contracts that have (potential) vulnerabilities and have an indication of how frequent a specific problem is. It can also be used to compare analysis tools in terms of metrics such as performance.

2.3.1 SB$^{\text{CURATED}}$: A Dataset of 69 Vulnerable Smart Contracts.

Goal. Our objective in constructing this dataset is to collect a set of Solidity smart contracts with known vulnerabilities, from deployed contracts in the Ethereum network to examples provided to illustrate vulnerabilities, that can serve as a dataset suite for research in the security analysis of Solidity smart contracts. We use the taxonomy presented in the DASP[3] to describe vulnerabilities of Ethereum smart contracts (see Categories in Table 3). Each collected contract is classified in one of the ten categories. We also manually tagged the lines that contain the vulnerability. This classification allows for new smart contract analysis tools to be easily evaluated.

Collection Methodology. This dataset has been created by collecting contracts from three different sources: 1. GitHub repositories, 2. blog posts that analyze contracts, and 3. the Ethereum network. 80% of the contracts were collected from GitHub repositories. We

[2] Ether is the cryptocurrency of Ethereum.

[3] Decentralized Application Security Project (or DASP): https://dasp.co

Table 3: Categories of vulnerabilities available in the dataset SB$^{\text{CURATED}}$. For each category, we provide a description, the level at which the attack can be mitigated, the number of contracts available within that category, and the total number of lines of code in the contracts of that category (computed using cloc 1.82).

Category	Description	Level	Contracts	Vulns	LoC
Access Control	Failure to use function modifiers or use of tx.origin	Solidity	17	19	899
Arithmetic	Integer over/underflows	Solidity	14	22	295
Bad Randomness	Malicious miner biases the outcome	Blockchain	8	31	1,079
Denial of service	The contract is overwhelmed with time-consuming computations	Solidity	6	7	177
Front running	Two dependent transactions that invoke the same contract are included in one block	Blockchain	4	7	137
Reentrancy	Reentrant function calls make a contract to behave in an unexpected way	Solidity	7	8	778
Short addresses	EVM itself accepts incorrectly padded arguments	EVM	1	1	18
Time manipulation	The timestamp of the block is manipulated by the miner	Blockchain	4	5	76
Unchecked low level calls	call(), callcode(), delegatecall() or send() fails and it is not checked	Solidity	5	12	225
Unknown Unknowns	Vulnerabilities not identified in DASP 10	`N/A`	3	3	115
Total			69	115	3,799

identified GitHub repositories that match relevant search queries (*'vulnerable smart contracts'*, *'smart contracts security'*, *'solidity vulnerabilities'*) and contain vulnerable smart contracts. We searched Google using the same queries. We found several repositories with vulnerable smart contracts such as *not-so-smart-contracts*[4] and *SWC Registry*[5]. The latter is a classification scheme for security weaknesses in Ethereum smart contracts that is referenced in Mythril's GitHub repository. We also extracted vulnerabilities that come from trusted entities in blog posts where smart contracts are audited, tested or discussed such as Positive.com[6] and Blockchain.unica[7]. And finally, we used Etherscan[8] to collect smart contracts that are deployed on the Ethereum network and are known to contain vulnerabilities (e.g. the original SmartBillions contract). Note that all the contracts were collected from trusted entities in the field. We also ensure the traceability of each contract by providing the URL from which they were taken and its author, where possible.

Dataset Statistics. The dataset contains 69 contracts and 115 tagged vulnerabilities, divided into ten categories of vulnerabilities. Table 3 presents information about the 69 contracts. Each line contains a category of vulnerability. For each category, we provide a description, the level at which the attack can be mitigated, the number of contracts available within that category, and the total number of lines of code in the contracts of that category.

Dataset Availability. The dataset is available in the repository of SmartBugs [14]. The dataset is divided into ten folders named with the DASP categories, and the folders contain the contracts of that category. Moreover, the dataset contains the file `vulnerabilities.json` which contains the details of each vulnerable contract. It details the name, the origin URL, the path, and the lines and the category of the vulnerabilities.

2.3.2 SB^{WILD}: 47,518 Contracts from the Ethereum Blockchain.

Goal. The goal of this second dataset is to collect as many smart contracts as possible from the Ethereum blockchain, in order to have a representative picture of the practice and (potential) vulnerabilities that are present in the production environment.

Collection Methodology. The data collection for the second dataset follows a different strategy. In this dataset, we collect all the different contracts from the Ethereum blockchain. Etherscan allows downloading the source code of a contract if you know its address. Therefore, we firstly use Google BigQuery [29] to collect the Ethereum contract addresses that have at least one transaction. We used the following BigQuery request to select all the contract addresses and count the number of transactions that are associated with each contract[9]:

```
SELECT contracts.address, COUNT(1) AS tx_count
  FROM `ethereum_blockchain.contracts` AS contracts
  JOIN `ethereum_blockchain.transactions` AS transactions
        ON (transactions.to_address = contracts.address)
  GROUP BY contracts.address
  ORDER BY tx_count DESC
```

After collecting all the contract addresses, we used Etherscan and its API to retrieve the source code associated with an address. However, Etherscan does not have the source code for every contract. Therefore, at the end of this step, we obtained a Solidity file for each contract that has its source code available in the Etherscan platform.

The final step was to filter the set of contracts to remove duplicates. Indeed, we observe that 95% of the available Solidity contracts are duplicates. We consider that two contracts are duplicates when the MD5 checksums of the two source files are identical after removing all the spaces and tabulations.

Dataset Statistics. Table 4 presents the statistics of this dataset. The query on Google BigQuery retrieved 2,263,096 smart contract addresses. We then requested Etherscan for the Solidity source code of those contracts, and we obtained 972,975 Solidity files. This means that 1,290,074 of the contracts do not have an associated source file in Etherscan. The filtering process of removing duplicate contracts resulted in 47,518 unique contracts (a total of 9,693,457 lines). According to Etherscan, 47 contracts that we requested do not exist (we labelled those as Unaccessible).

[4] not-so-smart-contracts: https://github.com/crytic/not-so-smart-contracts
[5] SWC Registry: https://smartcontractsecurity.github.io/SWC-registry
[6] Positive.com: https://blog.positive.com
[7] Blockchain.unica: http://blockchain.unica.it/projects/ethereum-survey/
[8] Etherscan: https://etherscan.io
[9] The query is also available at the following URL: https://bigquery.cloud.google.com/savedquery/281902325312:47fd9afda3f8495184d98db6ae36a40c

Table 4: Statistics on the collection of Solidity smart contracts from the Ethereum blockchain.

Solidity source not available	1,290,074
Solidity source available	972,975
Unaccessible	47
Total	2,263,096
Unique Solidity Contracts	47,518
LOC	9,693,457

Dataset Availability. The dataset is available on GitHub [15]. The dataset contains the Solidity source code of each of the 47,518 contracts. The contracts are named with the address of the contract. We also attached with this dataset additional information in order to use this dataset for other types of studies. It contains: • the name of the contract; • the Solidity version that has been used to compile the contract; • the addresses of the duplicated contracts; • the number of transactions associated with the contract; • the size of the contract in lines of Solidity code; • the date of the last transactions for the 2,263,096 contracts; • the date of creation for the 2,263,096 contracts; and • the Ethereum balance of 972,975 contracts that have their source code available.

2.4 The Execution Framework: SmartBugs

We developed SmartBugs, an execution framework aiming at simplifying the execution of analysis tools on datasets of smart contracts. SmartBugs has the following features: • A plugin system to easily add new analysis tools, based on Docker images; • Parallel execution of the tools to speed up the execution time; • An output mechanism that normalizes the way the tools are outputting the results, and simplify the process of the output across tools.

SmartBugs currently supports 9 tools (see Section 2.2).

2.4.1 Architecture. SmartBugs is composed of five main parts: (1) The first consists of the command-line interface to use SmartBugs (see Section 2.4.2). (2) The second part contains the tool plugins. Each tool plugin contains the configuration of the tools. The configuration contains the name of the Docker image, the name of the tool, the command line to run the tool, the description of the tool, and the location of the output of results. (3) The Docker images that are stored on Docker Hub. We use Docker images of the tools when a Docker image is already available; otherwise, we create our own image (all Docker images are publicly available on Docker Hub, including our own). (4) The datasets of smart contracts (see Section 2.3). (5) The SmartBugs' runner puts all the parts of SmartBugs together to execute the analysis tools on the smart contracts.

2.4.2 Dataset Interface Details. SmartBugs provides a command-line interface that simplifies the execution of the smart contract analysis tools. It takes a set of tool names and a path to Solidity files to analyze and produces two files per execution: 1) a `result.log` file that contains the stdout of the execution and 2) a `result.json` file that contains the results of the analysis in a parsable format. Moreover, we provide scripts that process those outputs and render them in readable tables such as the one presented in this paper.

2.5 Data Collection and Analysis

To answer our research questions, we used SmartBugs to execute the 9 tools on the two datasets described in Section 2.3. We collected the output and used it for further analysis. In this section, we describe the setup of the tools (Section 2.5.1) and their execution (Section 2.5.2).

2.5.1 Tools' Setup. For this experiment, we set the time budget to 30 minutes per analysis. In order to identify a suitable time budget for one execution of one tool over one contract, we first executed all the tools on SB$^{\text{CURATED}}$ dataset. We then selected a time budget that is higher than the average execution time (one minute and 44 seconds). If the time budget is spent, we stop the execution and collect the partial results of the execution. During the execution of our experiment, Manticore was the only tool that faced timeouts.

2.5.2 Large-scale Execution. To our knowledge, we present the largest experimental study on smart contract analysis, both in the number of tools and in execution time. In total, we executed 9 analysis tools on 47,518 contracts. This represents 428,337 analyzes, which took approximately 564 days and 3 hours of combined execution, more than a year of continuous execution. We used two cloud providers to rent the servers required for this experiment. The first provider was Scaleway[10], where we used three servers with 32 vCPUs with 128 GB of RAM. We added a budget of 500 €, and we spent 474.99 €.

The second provider was Google Cloud[11], where we also used three servers with 32 vCPUs with 30GB of RAM. We spent 1038.46 € with Google Cloud. In total, we spent 1513.45 € to execute the experiments discussed in this paper. We used two cloud providers due to administrative restrictions on our budget line. We were initially targeting Scaleway because it is cheaper than Google Cloud, but we were not able to spend more than 500 € with this provider. All the logs and the raw results of the analysis are available at [13].

3 RESULTS

The results of our empirical study, as well as the answers to our research questions, are presented in this section.

3.1 Precision of the Analysis Tools (RQ1)

To answer the first research question, we used SB$^{\text{CURATED}}$, the dataset of 69 contracts described in Section 2.3.1. Since each contract of this dataset is categorized in one of the ten DASP categories, we can compute the ability of the 9 tools in detecting the vulnerabilities present in the 69 contracts. The methodology that we followed to answer this research question was the following: (1) We executed the 9 tools on the 69 contracts. The result of this execution is available on GitHub [13]. (2) We extracted all the vulnerabilities that were detected by the tools into a JSON file. (3) We mapped the detected vulnerabilities to a category of vulnerabilities (see Table 3). To achieve this task, we manually annotated all the vulnerability types that have been detected into one of the ten DASP categories. For example, Oyente detects a vulnerability called `Integer Overflow` that we link to the category *Arithmetic*. In total, we identify 141 vulnerability types, and 97 of them have been tagged in one of the

[10] Scaleway: https://www.scaleway.com
[11] Google Cloud: https://cloud.google.com

Table 5: Vulnerabilities identified per category by each tool. The number of vulnerabilities identified by a single tool is shown in brackets.

Category	HoneyBadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0/19 0%	0/19 0%	4/19 21%	4/19 21%	0/19 0%	0/19 0%	0/19 0%	4/19 21% (1)	2/19 11%	5/19 26%
Arithmetic	0/22 0%	0/22 0%	4/22 18%	15/22 68%	11/22 50% (2)	12/22 55% (2)	0/22 0%	0/22 0%	1/22 5%	19/22 86%
Denial Service	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/ 7 0%
Front Running	0/7 0%	0/7 0%	0/7 0%	2/7 29%	0/7 0%	0/7 0%	2/7 29%	0/7 0%	0/7 0%	2/ 7 29%
Reentrancy	0/8 0%	0/8 0%	2/8 25%	5/8 62%	5/8 62%	5/8 62%	5/8 62%	7/8 88% (2)	5/8 62%	7/ 8 88%
Time Manipulation	0/5 0%	0/5 0%	1/5 20%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	2/5 40% (1)	1/5 20% (1)	3/ 5 60%
Unchecked Low Calls	0/12 0%	0/12 0%	2/12 17%	5/12 42% (1)	0/12 0%	0/12 0%	3/12 25%	4/12 33% (3)	4/12 33% (1)	9/12 75%
Other	2/3 67%	0/3 0%	0/3 0%	0/3 0%	0/3 0%	0/3 0%	0/3 0%	3/3 100% (1)	0/3 0%	3/ 3 100%
Total	2/115 2%	0/115 0%	13/115 11%	31/115 27%	16/115 14%	17/115 15%	10/115 9%	20/115 17%	13/115 11%	48/115 42%

Table 6: Total number of detected vulnerabilities by each tool, including vulnerabilities not tagged in the dataset.

Category	HoneyBadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0	10	28	24	0	0	6	20	3	91
Arithmetic	0	0	11	92	62	69	0	0	23	257
Denial of Service	0	0	0	0	27	11	0	2	19	59
Front Running	0	0	0	21	0	0	55	0	0	76
Reentrancy	0	0	4	16	5	5	32	15	7	84
Time Manipulation	0	0	4	0	4	5	0	5	2	20
Unchecked Low Level Calls	0	0	4	30	0	0	21	13	14	82
Unknown Unknowns	5	2	25	32	0	0	0	28	8	100
Total	5	12	76	215	98	90	114	83	76	769

ten categories. The remaining 44 do not fit the DASP taxonomy (for example, some tools warn about the use of inline assembly, which is considered a bad practice but does not necessarily lead to vulnerable contracts)[12] (4) At this point, we were able to identify which vulnerabilities the tools detect. Unfortunately, we found out that none of the 9 tools were able to detect vulnerabilities of the categories *Bad Randomness* and *Short Addresses*. This is unsurprising: it is expected that tools do not detect vulnerabilities of certain categories, since they are not designed to identify all types of vulnerabilities. Despite not seeing this as a limitation of the studied tools, we argue that our analysis gives insight on opportunities to improve them, since it provides an overview on how tools perform with respect to the taxonomy used.

The results of this first study are presented in Table 5 and Table 6. The first table presents the number of known vulnerabilities that have been identified. A vulnerability is considered as identified when a tool detects a vulnerability of a specific category at a specific line, and it matches the vulnerability that has been annotated in the dataset. Each row of Table 5 represents a vulnerability category, and each cell presents the number of vulnerabilities where the tool detects a vulnerability of this category. Some cells in Table 5 have numbers enclosed in brackets: these denote the number of vulnerabilities identified by a single tool. This table summarizes the strengths and weaknesses of the current state of the art of smart contract analysis tools. It shows that the tools can accurately detect vulnerabilities of the categories *Arithmetic*, *Reentrancy*, *Time manipulation*, *Unchecked Low Level Calls*, and *Unknown Unknowns*. With respect to the category *Unknown Unknowns*, the tools detected vulnerabilities such as the presence of uninitialized data and the possibility of locking down Ether. However, they were not accurate in detecting vulnerabilities of the categories *Access Control*, *Denial of service*, and *Front running*. The categories *Bad Randomness* and *Short Addresses* are not listed, since none of the tools are able to detect vulnerabilities of these types. This shows that there is still room for improvement and, potentially, for new approaches to detect vulnerabilities of the ten DASP categories.

Table 5 also shows that the tools offer distinct accuracies. Indeed, the tool *Mythril* has the best accuracy among the 9 tools. *Mythril* detects 27% of all the vulnerabilities when the average of all tools is 12%. The ranking of the tools is comparable to the one observed by Parizi *et al.* [34]. However, the average accuracy is lower on our benchmark SB$^{\text{CURATED}}$. Moreover, *Mythril*, *Manticore*, *Slither*, and *Smartcheck* are the tools that detect the largest number of different categories (5 categories). We can also see that *Slither* is the tool that uniquely identifies more vulnerabilities (8 vulnerabilities across 5 categories). Despite its good results, *Mythril* is not powerful enough to replace all the tools: by combining the detection abilities of all the tools, we succeed to detect 42% of all the vulnerabilities. However, depending on the available computing power, it might not be realistic to combine all the tools.

Therefore, we suggest the combination of *Mythril* and *Slither*. This combination detects 42 (37%) unique vulnerabilities. This combination offers a good balance between performance and execution cost. This combination is the best possible combination by a considerable margin. The second best combination, *Mythrill* and *Oyente*, only succeeds to detect 33 (29%) of all the vulnerabilities.

We now consider all the vulnerability detections and not only the ones that have been tagged in SB$^{\text{CURATED}}$. Table 6 presents the total

[12] The mapping that we created is available at: https://github.com/smartbugs/smartbugs/wiki/Vulnerabilities-mapping.

number of vulnerabilities detected by the tools. This table allows the comparison of the total number of detected vulnerabilities with the number of detected known vulnerabilities shown in Table 5. The tools that are performing the best are also producing much more warnings (i.e., their output is more *noisy*), making it difficult for a developer to exploit their results.

Answer to RQ1. What is the accuracy of current analysis tools in detecting vulnerabilities on Solidity smart contracts? By combining the 9 tools together, they are only able to detect 42% of all the vulnerabilities. This shows that there is still room to improve the accuracy of the current approaches to detect vulnerabilities in smart contracts. We observe that the tools underperform to detect vulnerabilities in the following three categories: *Access Control*, *Denial of service*, and *Front running*. They are unable to detect by design vulnerabilities from *Bad Randomness* and *Short Addresses* categories. We also observe that *Mythril* outperforms the other tools by the number of detected vulnerabilities (31/115, 27%) and by the number of vulnerability categories that it targets (5/9 categories). The combination of *Mythril* and *Slither* allows detecting a total of 42/115 (37%) vulnerabilities, which is the best trade-off between accuracy and execution costs.

3.2 Vulnerabilities in Production Smart Contracts (RQ2)

To answer the second research question, we analyzed the ability of the 9 selected tools to detect vulnerabilities in the contracts from the dataset SB^{WILD} (described in Section 2.3.2). We followed the same methodology as in the previous research question, however, for SB^{WILD}, we do not have an oracle to identify the vulnerabilities.

Table 7 presents the results of executing the 9 tools on the 47,518 contracts. It shows that the 9 tools are able to detect eight different categories of vulnerabilities. Note that the vulnerabilities detected by *HoneyBadger* are contracts that look vulnerable but are not. They are designed to look vulnerable in order to steal Ether from people that tries to exploit the vulnerability. In total, 44,589 contracts (93%) have at least one vulnerability detected by one of the 9 tools.

Such a high number of vulnerable contracts suggests the presence of a considerable number of false positives. *Oyente* is the approach that identifies the highest number of contracts as vulnerable (73%), mostly due to vulnerabilities in the *Arithmetic* category. This observation is coherent with the observation of Parizi *et al.* [34], since they determine that *Oyente* has the highest number of false positives when compared to *Mythril*, *Securify*, and *Smartcheck*.

Since we observed a potentially large number of false positives, we analyzed to what extent the tools agree in vulnerabilities they flag. The hypothesis is that if a vulnerability is identified exclusively by a single tool, the probability of it being a false positive increases. Figure 1 presents the results of this analysis. This figure shows the proportion of detected vulnerabilities that have been identified exclusively by one tool alone, two tools, three tools, and finally by four or more tools. *HoneyBadger* has a peculiar, but useful role: if *HoneyBadger* detects a vulnerability, it actually means that the vulnerability does not exist. So, consensus with *HoneyBadger* suggests the presence of false positives.

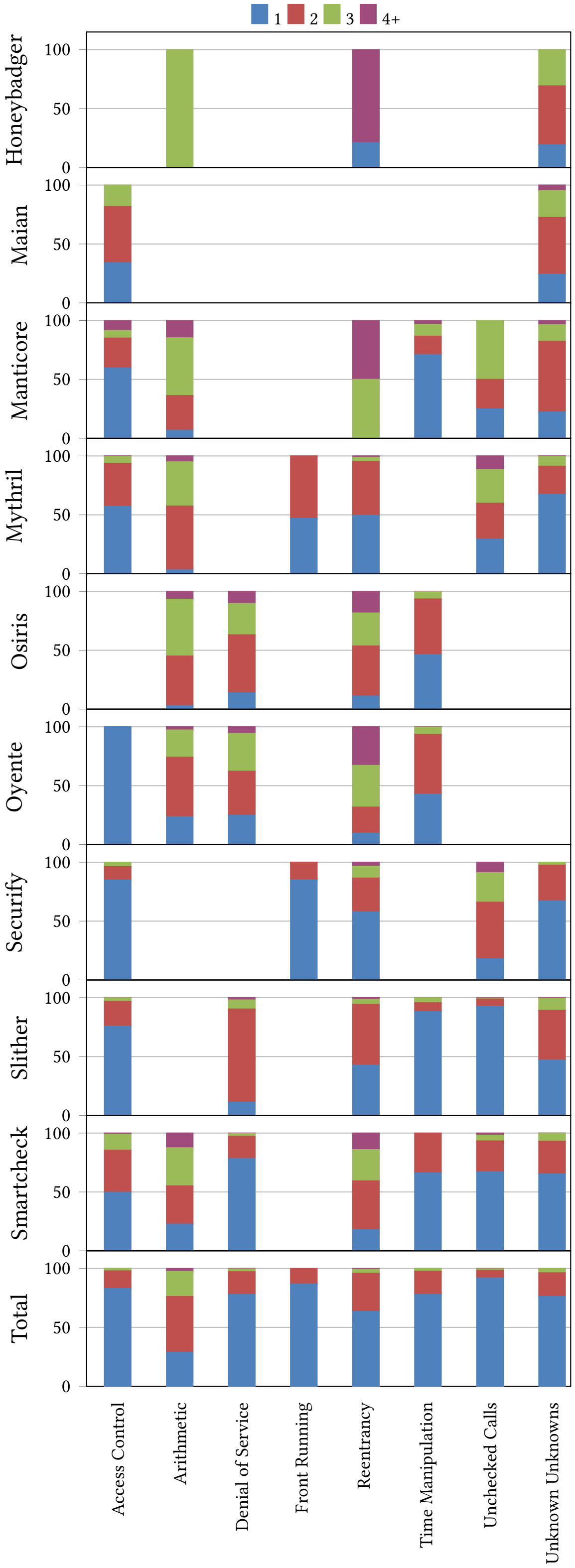

Figure 1: Proportion of vulnerabilities identified by exactly one (1), two (2) or three (3) tools, and by four tools or more (4+).

Table 7: The total number of contracts that have at least one vulnerability (analysis of 47,518 contracts).

Category	HoneyBadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0 0%	44 0%	47 0%	1,076 2%	0 0%	2 0%	614 1%	2,356 4%	384 0%	3,801 8%
Arithmetic	1 0%	0 0%	102 0%	18,515 39%	13,922 29%	34,306 72%	0 0%	0 0%	7,430 15%	37,597 79%
Denial of Service	0 0%	0 0%	0 0%	0 0%	485 1%	880 1%	0 0%	2,555 5%	11,621 24%	12,419 26%
Front Running	0 0%	0 0%	0 0%	2,015 4%	0 0%	0 0%	7,217 15%	0 0%	0 0%	8,161 17%
Reentrancy	19 0%	0 0%	2 0%	8,454 17%	496 1%	308 0%	2,033 4%	8,764 18%	847 1%	14,747 31%
Time Manipulation	0 0%	0 0%	90 0%	0 0%	1,470 3%	1,452 3%	0 0%	1,988 4%	68 0%	4,069 8%
Unchecked Low Calls	0 0%	0 0%	4 0%	443 0%	0 0%	0 0%	592 1%	12,199 25%	2,867 6%	14,656 30%
Unknown Unknows	26 0%	135 0%	1,032 2%	11,126 23%	0 0%	0 0%	561 1%	9,133 19%	14,113 29%	28,355 59%
Total	46 0%	179 0%	1,203 2%	22,994 48%	14,665 30%	34,764 73%	8,781 18%	22,269 46%	24,906 52%	44,589 93%

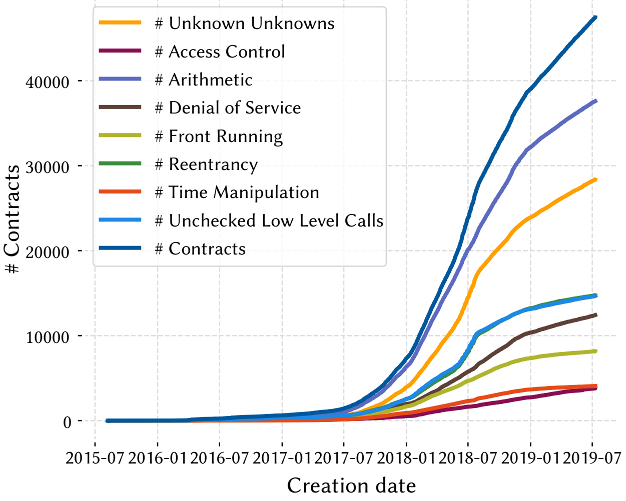

Figure 2: Evolution of number of vulnerabilities over time.

It is clear from the figure that the large majority of the vulnerabilities have been detected by one tool only. One can observe that there are 71.25% of the *Arithmetic* vulnerabilities found by more than one tool. It is also the category with the highest consensus between four and more tools: 937 contracts are flagged as having an *Arithmetic* vulnerability with a consensus of more than three tools. It is followed by the *Reentrancy* category with 133 contracts receiving a consensus of four tools or more. These results suggest that combining several of these tools may yield more accurate results, with fewer false positives and negatives.

The tool *HoneyBadger* is different: instead of detecting vulnerabilities, it detects malicious contracts that try to imitate vulnerable contracts in order to attract transactions to their honeypots. Therefore, when *HoneyBadger* is detecting a *Reentrancy* vulnerability, it means that the contract looks vulnerable to *Reentrancy* but it is not. Figure 1 shows that 15 contracts identified by *HoneyBadger* with vulnerabilities of type *Reentrancy* have been detected by three other tools as *Reentrancy* vulnerable.

We also analyzed the evolution of the vulnerabilities over time. Figure 2 presents the evolution of the number of vulnerabilities by category. It firstly shows that the total number of unique contracts started to increase exponentially at the end of 2017 when Ether was at its highest value. Secondly, we can observe two main groups of curves. The first one contains the categories *Arithmetic* and *Unknown Unknowns*. These two categories follow the curve of the total number of contracts. The second group contains the other categories. The growing number of vulnerable contracts seems to slow down from July 2018. Finally, this figure shows that the evolution of categories *Reentrancy* and *Unchecked Low Level Calls* is extremely similar (the green line of *Reentrancy* is also hidden by the blue line of *Unchecked Low Level Calls*). This suggests a correlation between vulnerabilities in these two categories.

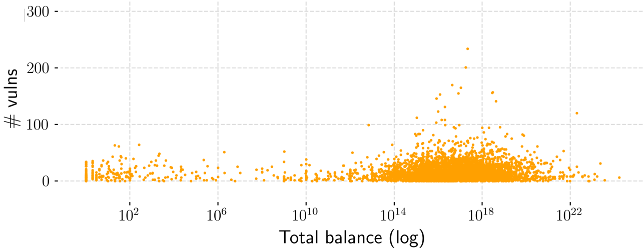

Figure 3: Correlation between the number of vulnerabilities and balance in Wei (one Ether is 10^{18} Wei).

And lastly, Figure 3 presents the correlation between the number of vulnerabilities and the balance of the contracts. It shows that the contracts that have a balance between 10^{14} Wei and 10^{20} Wei have more vulnerabilities than other contracts. Hence, the richest and the *middle class* seem to be less impacted. Per category, we have not observed any significant differences worth reporting.

Answer to RQ2. How many vulnerabilities are present in the Ethereum blockchain? The 9 tools identify vulnerabilities in 93% of the contracts, which suggests a high number of false positives. *Oyente*, alone, detects vulnerabilities in 73% of the contracts. By combining the tools to create a consensus, we observe that only a few number of vulnerabilities received a consensus of four or more tools: 937 for *Arithmetic* and 133 for *Reentrancy*.

3.3 Execution Time of the Analysis Tools (RQ3)

In this section, we present the execution time required by the tools to analyze the 47,518 of the SB$^{\text{WILD}}$ dataset (see Section 2.3.2). In order to measure the time of the execution, we recorded for each individual analysis when it started and when it ended. The duration of the analysis is the difference between the starting time and the ending time. An individual execution is composed of the following steps: 1) start the Docker image and bind the contract to the Docker instance; 2) clean the Docker container; and 3) parse and output the logs to the results folder.

Table 8 presents the average and total times used by each tool. The average execution time is per contract. It considers the execution of the tool on a contract, including compilation, construction of IR/graphs, analysis and parsing the results. In the table, we can observe three different groups of execution time: the tools that take a few seconds to execute, the tools that take a few minutes, and *Manticore* that takes 24 minutes. *Oyente*, *Osiris*, *Slither*, *Smartcheck*, and *Solhint* are much faster tools that take between 5 and 30 seconds on average to analyze a smart contract. *HoneyBadger*, *Maian*, *Mythril*, and *Securify* are slower and take between 1m24s and 6m37s to execute. Finally, Manticore takes 24m28s. The difference in execution time between the tools is dependent on the technique that each tool uses. Pure static analysis tools such as *Smartcheck* and *Slither* are fast since they do not need to compile nor execute contracts to identify vulnerabilities and bad practices.

Securify, *Maian*, *Mythril*, and *Manticore* analyze the EVM bytecode of the contracts. It means that those tools require the contract to be compiled before doing the analysis. The additional compilation step slows down the analysis. *Manticore* is the slowest of all the tools because this tool only analyzes an internal contract at a time (Solidity source files can contain an arbitrary number of contract definitions). Consequently, this tool has the major drawback of having the compilation overhead for each internal contract that it analyzes.

The average execution time does not reflect the complete picture of the performance of a tool. For example, *Maian* and *Manticore* use several processes, and *Maian* uses up to 16GB of RAM. Consequently, *Maian* and *Manticore* are difficult to parallelize. We were able to run only four, and ten parallel executions for respectively *Maian* and *Manticore* on a 32-core server with 30GB of RAM. This also explains why we were not able to execute those two tools on the complete dataset of 47,518 smart contracts.

Interestingly, the slowest tools do not have better accuracy (see Section 3.1). *Mythril*, for example, which has the best accuracy according to our evaluation, takes on average of 1m24s to analyze a contract. It is much faster than *Manticore* that only has an accuracy of 11% compared to the 27% of *Mythril*.

The execution time of *Maian* is surprising compared to the results that have been presented in the *Maian* paper [32]. Indeed, the authors claimed that it takes on average 10 seconds to analyze a contract while we observe that it takes 5m16s in our experiment on similar hardware. The difference in execution times can potentially be explained by the difference of input uses in the two experiments. We use the source code of the contract as input, and *Maian*'s authors use the bytecode. The overhead for the compilation of the contract seems to be the major cost of execution for this tool.

Table 8: Average execution time for each tool.

#	Tools	Execution time	
		Average	Total
1	Honeybadger	0:01:38	23 days, 13:40:00
2	Maian	0:05:16	49 days, 10:06:15
3	Manticore	0:24:28	184 days, 01:59:02
4	Mythril	0:01:24	46 days, 07:46:55
5	Osiris	0:00:34	18 days, 10:19:01
6	Oyente	0:00:30	16 days, 04:50:11
7	Securify	0:06:37	217 days, 22:46:26
8	Slither	0:00:05	2 days, 15:09:36
9	Smartcheck	0:00:10	5 days, 12:33:14
Total		0:04:31	564 days, 3:10:39

Answer to RQ3. How long do the tools require to analyze the smart contracts? On average, the tools take 4m31s to analyze one contract. However, the execution time largely varies between the tools. *Slither* is the fastest tool and takes on average only 5 seconds to analyze a contract. *Manticore* is the slowest tool. It takes on average 24m28s to analyze a contract. We also observe that the execution speed is not the only factor that impacts the performance of the tools. *Securify* took more time to execute than *Maian*, but *Securify* can easily be parallelized and therefore analyze the 47,518 contracts much faster than *Maian*. Finally, we have not observed a correlation between accuracy and execution time.

4 DISCUSSION

We discuss the practical implications of our findings from the previous section, as well as outline the potential threats to their validity.

4.1 Practical Implications and Challenges

Despite the advances in automatic analysis tools of smart contracts during the last couple of years, the practical implication our study highlights is that there remain several open challenges to be tackled by future work. We identify four core challenges: increasing and ensuring the quality of the analysis, extending the scope of problems addressed by these tools, integrating the analysis into the development process, and extending the current taxonomy.

Quality: This challenge is about increasing the likelihood that a tool identifies real vulnerabilities, yielding close to zero false positives and false negatives. Our study demonstrates that this is far from being the case, and future work should be invested in improving the quality of the tools. Addressing this challenge is perhaps an important step toward real-life adoption of these tools and techniques.

Scope: Although there might not be a technique that finds all sorts of vulnerabilities, this challenge is about further extending existing techniques so that more real vulnerabilities can be found. In the previous section, we briefly discussed a potential way to address this challenge: crafting a novel technique combining complementary tools. Combining static with dynamic analysis might also be an interesting avenue for future work.

Development process: This challenge is about integrating these tools into the development process, thus contributing to real-life adoption. To ease the interaction of developers with these tools, hence making them useful during the development life-cycle, the following could bring added value: integration with other orthogonal techniques (such as bug detection tools, dynamic analysis techniques, and generic linters), integration with popular IDEs, interactive reports (e.g., highlight vulnerable code), and explainable warnings.

Taxonomy: Another practical implication of our work is that the current state-of-the-art set of 10 categories in DASP10 does not seem to be comprehensive enough to cover all vulnerabilities that affect smart contracts deployed in Ethereum. DASP10 includes the category *Unknown Unknowns*, because as the creators of the DASP taxonomy observed, *as long as investors decide to place large amounts of money on complex but lightly-audited code, we will continue to see new discoveries leading to dire consequences*. Our work sheds light on potential new categories that could extend DASP10, such as *Dependence on environment data* and *Locked Ether*. The latter could include not only the cases where Ether is locked indefinitely, but also cases of *Honeypots* as defined by Torres *et al.* [41] (since Ether becomes locked, except for the attacker). Our findings suggest new categories comparable to those proposed in the recent survey by Chen *et al.* [7] (available online on August 13, 2019).

4.2 Threats to Validity

A potential threat to the internal validity is that, due to the complexity of the SmartBugs framework, there may remain an implementation bug somewhere in the codebase. We extensively tested the framework to mitigate this risk. Furthermore, the framework and the raw data are publicly available for other researchers and potential users to check the validity of the results.

A potential threat to the external validity is related to the fact that the set of smart contracts we have considered in this study may not be an accurate representation of the set of vulnerabilities that can happen during development. We attempt to reduce the selection bias by leveraging a large collection of real, reproducible smart contracts. Another potential threat is that we may have missed a tool or failed to reproduce a tool that excels all other tools. To mitigate this risk, we contacted the authors of the tools if no source code was found. We also aim to reduce threats to external validity and ensure the reproducibility of our evaluation by providing the source of our instrumentation tool, the scripts used to run the evaluation, and all data gathered.

A potential threat to the construct validity relates to the fact that we needed to manually label the vulnerable smart contracts into one of the DASP categories, and these could be mislabeled. This risk was mitigated as follows: all authors labeled the contracts, and then disagreements were discussed to reach a consensus. Another potential threat to the validity is the timeout used for the analysis: 30 minutes. This threat was mitigated by executing all the tools on the SB$^{\text{CURATED}}$ dataset.

The analysis based on the dataset SB$^{\text{WILD}}$ is valuable for it gives a sense of the potential noise that tools might generate when analysing contracts. However, with such a large dataset, the discussion on the number of false positives is challenging and there is a risk that too many false positives are identified. To mitigate this risk, we decided to use consensus as a proxy: the hypothesis is that the greater the number of tools identifying a vulnerability, the more likely that vulnerability is a true positive.

5 RELATED WORK

As discussed in Section 2.2, there are several automated analysis tools available. Notwithstanding, despite the recent increase interest in the analysis of smart contracts, to the best of our knowledge, our work is the first systematic comparison of recently proposed techniques to better understand their real capabilities.

Datasets and Repositories: Reproducibility is enabled by a benchmark containing smart contracts that other researchers can use *off-the-shelf*. There are just a few repositories of smart contracts available to the research community, such as *VeriSmartBench*[13], *evm-analyzer-benchmark-suite*[14], *EthBench*[15], *smart-contract-benchmark*[16], and *not-so-smart-contracts*[17]. These, however, are essentially collections of contracts and are not designed to enable reproducibility nor to facilitate comparison of research. Our dataset SB$^{\text{CURATED}}$ offers a known vulnerability taxonomy, positioning itself as a reference dataset to the research community.

Empirical studies: Chen *et al.* [8] discussed an empirical study on code smells for Ethereum smart contracts. Based on posts from Stack Exchange and real-world smart contracts, they defined 20 distinct code smells for smart contracts and categorized them into security, architecture, and usability issues. Furthermore, they manually labeled a dataset of smart contracts.[18] Pinna *et al.* [36] performed a comprehensive empirical study of smart contracts deployed on the Ethereum blockchain with the objective to provide an overview of smart contracts features, such as type of transactions, the role of the development community, and the source code characteristics. Parizi *et al.* [34] carried out an experimental assessment of static smart contracts security testing tools. They tested Mythril, Oyente, Securify, and Smartcheck on ten real-world smart contracts. Concerning the accuracy of the tools, Mythril was found to be the most accurate. Our results corroborate the findings, but in a more systematic and comprehensive manner.

Execution Frameworks: Execution frameworks to simplify and automate the execution of smart contract analysis tools are scarce. Solhydra [10] is a CLI tool to analyze Solidity smart contracts with several static analysis tools. It generates a report with results of the tool analysis. Unlike SmartBugs, which was designed to ease the addition of new analysis tools, Solhydra does not offer this flexibility. Furthermore, Solhydra has not been updated in more than a year.

6 CONCLUSION

In this paper, we presented an empirical evaluation of 9 automated analysis tools on 69 annotated vulnerable contracts and on 47,518

[13] VeriSmartBench: https://github.com/soohoio/VeriSmartBench
[14] evm-analyzer-benchmark-suite: https://github.com/ConsenSys/evm-analyzer-benchmark-suite
[15] EthBench: https://github.com/seresistvanandras/EthBench
[16] smart-contract-benchmark: https://github.com/hrishioa/smart-contract-benchmark
[17] not-so-smart-contracts: https://github.com/crytic/not-so-smart-contracts
[18] CodeSmell: https://github.com/CodeSmell2019/CodeSmell

contracts taken from the Ethereum's network. The goal of this experiment was to obtain an overview of the current state of automated analysis tools for Ethereum smart contracts. During this empirical evaluation, we considered all available smart contracts analysis tools and all the available Ethereum contracts that have at least one transaction. We used the DASP10 category of smart contract vulnerabilities as the reference to classify vulnerabilities.

We found out that the current state of the art is not able to detect vulnerabilities from two categories of DASP10: *Bad Randomness* and *Short Addresses*. Also, the tools are only able to detect together 48/115 (42%) of the vulnerabilities from our SB$^{\text{CURATED}}$ dataset. *Mythril* is the tool that has the higher accuracy and is able to detect 31/115 (27%) of the vulnerabilities.

During the evaluation of the 9 tools on SB$^{\text{WILD}}$, we observe that 97% of the contracts are identified as vulnerable. This suggests a considerable number of false positives. *Oyente* plays an important role in this, since it detects vulnerabilities in 73% of the contracts, mostly due to *Arithmetic* vulnerabilities (72%). Finally, we observe that the tools (4 or more) succeed to find a consensus for 937 *Arithmetic* vulnerabilities and 133 *Reentrancy* vulnerabilities.

We argue that the execution framework and the two new datasets presented here are valuable assets for driving reproducible research in automated analysis of smart contracts.

ACKNOWLEDGMENTS

This work has been co-funded by the European Union's Horizon 2020 research and innovation programme under the QualiChain project, Grant Agreement No 822404 and supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under projects UIDB/50021/2020 and PTDC/CCI-COM/29300/2017.

REFERENCES

[1] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. 2018. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In *Automated Technology for Verification and Analysis*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 513–520.

[2] Shaun Azzopardi, Joshua Ellul, and Gordon J. Pace. 2018. Monitoring Smart Contracts: ContractLarva and Open Challenges Beyond. In *Runtime Verification*, Christian Colombo and Martin Leucker (Eds.). Springer International Publishing, Cham, 113–137.

[3] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. 2016. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, New York, NY, USA, 91–96.

[4] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A scalable security analysis framework for smart contracts. arXiv:1809.03981

[5] Vitalik Buterin et al. 2013. Ethereum white paper. *GitHub repository* 1, GitHub (2013), 22–23.

[6] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, and Zijiang Yang. 2018. sCompile: Critical path identification and analysis for smart contracts. arXiv:1808.00624

[7] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2019. A Survey on Ethereum Systems Security: Vulnerabilities, Attacks and Defenses. arXiv:1908.04507

[8] Jiachi Chen, Xin Xia, David Lo, John Grundy, Daniel Xiapu Luo, and Ting Chen. 2019. Domain Specific Code Smells in Smart Contracts. arXiv:arXiv:1905.01467

[9] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Klagenfurt, Austria, 442–446.

[10] Blockchain Company. 2018. Solhydra. https://github.com/BlockChainCompany/solhydra.

[11] Phil Daian. 2016. Analysis of the DAO exploit. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

[12] M. di Angelo and G. Salzer. 2019. A Survey of Tools for Analyzing Ethereum Smart Contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*. IEEE, Newark, CA, USA, USA, 69–78. https://doi.org/10.1109/DAPPCON.2019.00018

[13] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2019. SmartBugs execution results. https://github.com/smartbugs/smartbugs-results.

[14] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2019. SmartBugs repository. https://github.com/smartbugs/smartbugs.

[15] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2019. SmartBugs Wild dataset. https://github.com/smartbugs/smartbugs-wild.

[16] Josselin Feist, Gustavo Greico, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *Proceedings of the 2Nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB '19)*. IEEE Press, Piscataway, NJ, USA, 8–15. https://doi.org/10.1109/WETSEB.2019.00008

[17] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 116.

[18] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Principles of Security and Trust*, Lujo Bauer and Ralf Küsters (Eds.). Springer International Publishing, Cham, 243–269.

[19] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Principles of Security and Trust*, Lujo Bauer and Ralf Küsters (Eds.). Springer International Publishing, Cham, 243–269.

[20] Peter Hegedus. 2019. Towards analyzing the complexity landscape of solidity based ethereum smart contracts. *Technologies* 7, 1 (2019), 6.

[21] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. 2018. KEVM: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, Oxford, UK, 204–217.

[22] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. NDSS, San Diego, California, USA, 1–15.

[23] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1317–1333. https://www.usenix.org/conference/usenixsecurity18/presentation/krupp

[24] Shuvendu K Lahiri, Shuo Chen, Yuepeng Wang, and Isil Dillig. 2018. Formal Specification and Verification of Smart Contracts for Azure Blockchain. arXiv:1812.08829

[25] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, New York, NY, USA, 65–68.

[26] Han Liu, Chao Liu, Wenqi Zhao, Yu Jiang, and Jiaguang Sun. 2018. S-gram: towards semantic-aware security auditing for ethereum smart contracts. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 814–819.

[27] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, New York, NY, USA, 254–269.

[28] Anastasia Mavridou and Aron Laszka. 2018. Tool Demonstration: FSolidM for Designing Secure Ethereum Smart Contracts. In *Principles of Security and Trust*, Lujo Bauer and Ralf Küsters (Eds.). Springer International Publishing, Cham, 270–277.

[29] Evgeny Medvedev. 2018. Ethereum in BigQuery: a Public Dataset for smart contract analytics. https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics.

[30] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. arXiv:1907.03890

[31] Bernhard Mueller. 2018. Smashing ethereum smart contracts for fun and real profit. In *9th Annual HITB Security Conference (HITBSecConf)*. HITB, Amsterdam, Netherlands, 54.

[32] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, New York, NY, USA, 653–663.

[33] Robert Norvill, Beltran Borja Fiz Pontiveros, Radu State, and Andrea Cullen. 2018. Visual emulation for Ethereum's virtual machine. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE, Taipei, Taiwan, 1–4.

[34] Reza M. Parizi, Ali Dehghantanha, Kim-Kwang Raymond Choo, and Amritraj Singh. 2018. Empirical Vulnerability Analysis of Automated Smart Contracts

Security Testing on Blockchains. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering (CASCON '18)*. IBM Corp., Riverton, NJ, USA, 103–113. http://dl.acm.org/citation.cfm?id=3291291.3291303

[35] Daniel Perez and Benjamin Livshits. 2019. Smart Contract Vulnerabilities: Does Anyone Care? arXiv:1902.06710

[36] Andrea Pinna, Simona Ibba, Gavina Baralla, Roberto Tonelli, and Michele Marchesi. 2019. A Massive Analysis of Ethereum Smart Contracts Empirical Study and Code Metrics. *IEEE Access* 7 (2019), 78194–78213.

[37] Matt Suiche. 2017. The $280M Ethereum's Parity bug. https://blog.comae.io/the-280m-ethereums-bug-f28e5de43513.

[38] Matt Suiche. 2017. Porosity: A decompiler for blockchain-based smart contracts bytecode. *DEF con* 25 (2017), 11.

[39] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, Gothenburg, Sweden, Sweden, 9–16.

[40] Christof Ferreira Torres, Julian Schütte, et al. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, New York, NY, USA, 664–676.

[41] Christof Ferreira Torres, Mathis Steichen, and Radu State. 2019. The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1591–1607. https://www.usenix.org/conference/usenixsecurity19/presentation/ferreira

[42] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, USA, 67–82.

[43] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara. 2018. Security Assurance for Smart Contract. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, Paris, France, 1–5. https://doi.org/10.1109/NTMS.2018.8328743

[44] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. 2018. Erays: Reverse Engineering Ethereum's Opaque Smart Contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1371–1385. https://www.usenix.org/conference/usenixsecurity18/presentation/zhou