A Study of Static Analysis Tools for Ethereum Smart Contracts

António Pedro Cruz Monteiro*
Instituto Superior Técnico, Universidade de Lisboa
apedrocruz@tecnico.ulisboa.pt

***Abstract*—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. Smart contracts, computer programs executed on top of a blockchain, are at the core of this technology, for they allow the creation of new distributed applications. Millions of smart contracts have been deployed on Ethereum, a major blockchain platform for smart contracts, and although they are seen with great potential, smart contracts have been known to have several security problems. Recent attacks on smart contracts and critical vulnerabilities discovered show us the huge financial loss they can impose on the users and warn us for the necessity of having methods that lead to secure and reliable implementations. Over the last few years several static analysis tools have been developed specifically targeting Ethereum smart contracts. In this dissertation, we present a review of state-of-the-art static analysis tools and introduce SmartBugs, a new extendable execution framework, created to facilitate the integration and comparison between multiple analysis tools and the analysis of Ethereum smart contracts. SmartBugs includes two datasets with a total of 47,661 Solidity smart contracts. We use SmartBugs to perform an empirical evaluation of 7 state-of-the-art automated analysis tools using the two new datasets. Our study shows that Mythril is the most sensitive and Slither has the best precision. In addition, we present an extension that improves SmartCheck.**

***Index Terms*—Blockchain, Ethereum, Smart Contracts, Static Analysis, Vulnerabilities, Solidity**

I. Introduction

Blockchain technology and cryptocurrencies, introduced by Bitcoin [2], have gained and continue to gain attention and popularity. Blockchain, a shared, decentralized, cryptographically secure and immutable digital ledger maintained by nodes in a peer-to-peer network is considered to be one of the most prominent technologies introduced in this century. Cryptocurrencies, which have become a global phenomenon, use this data structure as a public ledger to record all valid transactions. Blockchain acts as public append-only database that keeps a permanent and immutable record of digital transactions without relying on trust to secure it, instead it is secured by a peer-to-peer network using a consensus algorithm.

One of the blockchain appeals is being a technology that promises secure distributed computations even in the absence of trusted third parties. These computations, called *smart contracts* — interactions between mutually distrusting entities, are automatically enforced by the consensus mechanism of the blockchain, using incentives and cryptography. These immutable pieces of code, when deployed on the blockchain, allow parties to manifest contract terms in the form of program code.

Ethereum [3], which was introduced with the proclaimed intention of being a platform aimed to support smart contracts, is the most famous network to deploy smart contracts, where they written in a Turing-complete language. Solidity is the most used language by developers to create smart contracts on Ethereum. While smart contracts can represent a great improvement to our society model, as they do not rely on gained trust, they have shown several major security vulnerabilities. In a study performed on nearly one million Ethereum smart contracts, *34,200* of them were flagged as vulnerable [4]. A different study showed that of *19,366* smart contracts analysed, also in Ethereum, *8,833* (around 46%) were flagged as vulnerable [5]. A need to review and improve smart contracts implementation is therefore needed and it is a key element to enable massive adoption of smart contracts technology.

A. *Objectives and Contributions*

We propose to explore and discuss smart contracts and their secure implementation using static analysis tools, which can be used to analyse the correctness of the code. The main objectives are to study analysers that verify Ethereum smart contracts and introduce new tools and datasets to facilitate the use of static analysis in research and smart contracts development.To achieve these objectives, we introduce a novel, extendable, and easy-to-use execution framework, SmartBugs, that simplifies research and execution of automated analysis tools for smart contracts. With this framework we also introduce a dataset of 143 annotated vulnerable Solidity smart contracts, SB$^{\text{CURATED}}$, setting the ground for fair comparisons and a reference for the research community to use for benchmarks on analyses tools.

We use SmartBugs to execute 7 state-of-the-art static analysis tools on two datasets, SB$^{\text{CURATED}}$ and SB$^{\text{WILD}}$, with a total of 47,587 contracts being analysed. The second dataset contains all available smart contracts in the Ethereum blockchain that have Solidity source code available on Etherscan[1] (a total

* Parts of this work were done in collaboration with INESC-ID researchers, as documented in [1].

[1]Etherscan: https://etherscan.io

of 47,518 contracts). We perform an empirical analysis of this 7 tools on this datasets and present a discussion of the results.

To conclude, we present an extension of SmartCheck, a state-of-the-art static analysis tool, improving its detection of vulnerabilities and adding the capability to detect a category of vulnerabilities that previous was not in the scope of detection of SmartCheck. Throughtout this work, we propose to answer the following research questions:

RQ1. What are the state-of-the-art analysis tools for Ethereum smart contracts?
RQ2. What is the precision of the state-of-the-art analysis tools in detecting vulnerabilities on Ethereum smart contracts?
RQ3. Which categories of vulnerabilities are most detected?
RQ4. How many vulnerable contracts are present in the Ethereum blockchain?
RQ5. How long do the tools require to analyse the smart contracts?
RQ6. How can we improve a state-of-the-art analysis tool?

Our main contributions can be summarized as follows:

- We review, explain and discuss vulnerabilities found on Ethereum smart contracts;
- We present an overview of state-of-the-art automated smart contract analysers focused on Ethereum smart contracts;
- We introduce SmartBugs, an extendable execution framework designed to facilitate the integration and comparison between multiple analysis tools and the analysis of Ethereum smart contracts;
- We present SB$^{\text{CURATED}}$, a dataset of 143 annotated vulnerable Solidity smart contracts with a defined vulnerability taxonomy as a reference dataset to the research community;
- We provide an analysis of the execution of 7 state-of-the-art analysis tools with SmartBugs on 47,587 smart contracts;
- We extend SmartCheck, a state-of-the-art static analysis tool, to improve detection of vulnerabilities.

II. Background

A. Blockchain

In 2008 Satashi Nakamoto proposed Bitcoin, *"a system for electronic transactions without relying on trust"* [2], presenting a decentralized peer-to-peer network using proof-of-work, a consensus protocol, as a solution to the double-spending[2] problem. In this scenario, bitcoin (BTC) represents a cryptographic currency that can be exchanged between untrusted parties and the key idea is that the exchange of bitcoins between these entities is registered in a append-only database maintained by a network of peers that can not be tampered. This append-only database is built by chaining multiple validated blocks that contain transactions and is referred as blockchain. Blockchains can be divided in two different types: *permissionless* and *permissioned*. Bitcoin is the first permissionless blockchain, meaning that everyone can participate and interact with the network. Permissioned blockchains, on the other hand, are more restricted and the participants are controlled.

B. Smart Contracts

Smart contracts are pieces of code, deployed on a Blockchain, where it is possible to codify agreements and trust relations. They are able to keep a record of the state, exchange digital assets, take user input and store data. Smart contracts are stored and executed by the network of nodes who update the state of the smart contract on the blockchain when consensus on the outcome of the execution is reached. Users can exchange value or data and interact with contracts by publishing signed transactions to the network. Smart contract execution sometimes requires heavy computational tasks and since they are performed by computers (nodes) in the network, each computational task has a cost, which is called *gas* in Ethereum. These execution fees represent the cost paid by the user to the miners to execute code.

C. Ethereum

Ethereum [3] was introduced in 2014 and was the first to introduce a Turing-complete language in a blockchain platform, creating a second generation of blockchains. It has its own cryptocurrency, called *Ether* (ETH), and smart contracts represent one of the main components of the platform. Ethereum, like Bitcoin, is a permissionless blockchain where any network participant is allowed to interact and perform transactions within the network, such as transfer *Ether* or interact with a smart contract deployed on the network.

Smart contracts in Ethereum are compiled into a stack-based bytecode language and then executed by the *Ethereum Virtual Machine* (EVM). EVM is a turing complete virtual machine[3], specific to Ethereum that processes every transaction. Every participating node in the network runs their own instance of the EVM making possible the execution of code in a trustless environment. *Solidity* is the most used language by developers to create smart contracts on Ethereum. It is an object-oriented, high-level JavaScript-like language and is compiled to EVM bytecode to be executed by the EVM. Solidity is statically typed, supports inheritance, libraries and, like many programming languages, it offers common features like control flow structures, types, functions or structs. Several examples of Solidity code are shown in the next section.

D. Static Analysis Tools

Our research started off by using the survey of Angelo *et al.* [6] and we extended their list of tools by searching the academic literature and the internet for other tools. Through our research we were able to find more eight tools that suited our discussion. We ended up with the 35 static analysis tools that are listed below in Table I.

[2]Double-spending is the result of successfully spending the same digital token more than once. It is a potential flaw in a digital cash systems in which the same single digital token can be spent more than once.

[3]EVM code can encode any computation that can be conceivably carried out, including infinite loops.

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

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

Not all the identified tools are well suited for our discussion. Only the tools that met the following five inclusion criteria are relevant to be discussed:

- *Criterion #1.* [Available and CLI] The tool is publicly available and supports a command-line interface (CLI).
- *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.
- *Criterion #5.* [Static Analysis] The tool only executes static analysis techniques to find security issues.

After inspecting all 35 analysis tools presented in Table I, Table III defines the 7 tools that meet the inclusion criteria outlined. Table II presents the excluded tools and which criteria they did not meet.

Inclusion criteria	Tools that violate criteria
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
Static Analysis (C5)	contractLarva, Echidna, MAIAN, Manticore, ReGuard

Table II: Excluded analysis tools based on our inclusion criteria.

Tools that meet criteria	HoneyBadger, Mythril, Osiris, Oyente, Securify, Slither, Smartcheck

Table III: Included analysis tools based on our inclusion criteria.

HoneyBadger [15] is an Oyente-based (see below) tool that employs symbolic execution and a set of heuristics to pinpoint honeypots in smart contracts. 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.

Mythril [19] analyses EVM bytecode and relies on concolic analysis, which is a hybrid analysis technique that performs symbolic execution along a concrete execution path, to find several types of security issues.

Osiris [20] extends Oyente to improve the detection of integer bugs. It combines symbolic execution and taint analysis, in order to accurately find integer bugs in Ethereum smart contract.

Oyente [5] is one of the first static analysis tool for Ethereum smart contracts and has been forked by several other projects, including *HoneyBadger* and *Osiris*. It runs symbolic execution on EVM bytecode to identify vulnerabilities. Determines which inputs cause which program branches to execute to find potential security vulnerabilities. Oyente works directly with EVM bytecode without access high level representation.

Securify [25] statically analyses 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 [26] 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 [27] is a static analysis tool that looks for vulnerability patterns and bad coding practices. It runs lexical and syntactical analysis on Solidity source code.

III. SmartBugs

There has been some effort from the research community to develop automated analysis tools that locate and eliminate vulnerabilities in smart contracts [5], [25], [27], [33]. 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., [34]). Furthermore, most of smart contracts static analysers are required to be installed, and some even require you to install dependencies they rely on. Others are not available for all OS's. Executing smart contract analysis with multiple tools, running and installing each one individually can be an excruciating process and consume unnecessary time. There is a need to ease the execution of static analysers and provide a simple interface where the user could analyse multiple contracts with multiple tools without requiring installation of any of the tools.

We present SmartBugs to address these problems. SmartBugs is an extensible, and easy to use execution framework that simplifies research on automated analysis techniques for smart contracts and executes these tools on the same execution environment. To be able to execute and compare automated analysis tools, hence setting the ground for fair comparisons, we provide SB$^{\text{CURATED}}$: a dataset of 143 manually annotated Solidity smart contracts that can be used to evaluate the precision of analysis tools. Developers can use it to evaluate their automated analysis tools, create benchmarks and easily compare them with state of the art, alternative tools. SmartBugs provides the possibility to easily integrate analysis tools, thus enabling reproducibility of the results for these tools. Once tools are added to SmartBugs, it is easy to compare their performance and accuracy.

SmartBugs has the following features:

- A plugin system to easily add new analysis tools, based on Docker images;
- A plugin system to easily add new named datasets;
- Bulk analysis with any number of tools available;
- CLI and WUI Dashboard;
- Parallel execution of the tools to speed up the execution time; *[Contribution]*
- A parser mechanism that normalizes the way the tools are outputting the results; *[Contribution]*

The last two items marked with *[Contribution]* resulted in external contributions done to SmartBugs by co-authors of Durieux et al. [1] to automate results in the context of the analysis discussed in Section V.

A. Availability

SmartBugs comes with 10 tools ready to be executed: HoneyBadger, Maian, Manticore, Mythril, Osiris, Oyente, Securify, Slither, SmartCheck, Solhint. However, throughout this work we only focus on the seven tools that met the criteria presented in Section II-D. We excluded Maian, Manticore and Solhint.

The framework and SB$^{\text{CURATED}}$ are available in GitHub as repositories of SmartBugs [4][5].

B. SB$^{\text{CURATED}}$*: A Dataset of 143 Vulnerable Smart Contracts*

All contracts and tools present in SmartBugs are publicly available. The collection methodology is explained in this section. The dataset follows the taxonomy of DASP 10 [35]. Our objective in constructing SB$^{\text{CURATED}}$ was to provide a reliable dataset with a collection of vulnerabilities designed to be reproducible, follow a known taxonomy and to serve as a reference dataset to the research community. The last category presented in DASP 10 is *Unknown Unknowns*, which, represents future and undiscovered vulnerabilities. This category is not useful in the context of mapping existent vulnerabilities, so we opted to map vulnerabilities that did not fit any other of the nine categories, i.e. categories that are not identified in DASP 10, in this category. For example, vulnerabilities such as uninitialized data and the possibility of locking down Ether are mapped to this category. This category, *Unknown Unknowns*, will be referred as *Other* to avoid confusion and simplify the description.

Table IV: Categories of vulnerabilities available in the dataset. (computed using cloc 1.82).

Category	**Contracts**	**Vulns**	**LoC**
Access Control	17	19	899
Arithmetic	14	24	304
Bad Randomness	8	31	1,079
Denial of service	6	7	177
Front running	4	7	137
Reentrancy	31	32	2,164
Short addresses	1	1	18
Time manipulation	5	7	100
Unchecked low level calls	53	60	4055
Other	3	3	115
Total	143	191	9,048

SB$^{\text{CURATED}}$ design was driven by the following objectives: *Reproducibility*, *Real-world relevance*, *Diversity* and *Custom filtering*.

SB$^{\text{CURATED}}$ was created by collecting smart contracts from three different sources: GitHub repositories, Blog posts that analyse contracts and the Ethereum network. Most of contracts were collected from GitHub repositories and the Ethereum network. We ensure the traceability of each contract by providing the URL from which they were taken and its author, where possible.

[4]SmartBugs (including SB$^{\text{CURATED}}$): https://github.com/smartbugs/smartbugs

[5]SmartBugs WUI: https://github.com/smartbugs/smartbugs-dashboard

1) SB$^{\text{CURATED}}$ *Statistics:* The dataset contains 143 contracts and 191 manually tagged vulnerabilities, divided into 10 categories of vulnerabilities. Table IV presents information about the 143 contracts. Each line contains a category of vulnerability. For each category, we provide the number of contracts available within that category and the total number of vulnerabilities and number of lines of code in the contracts of that category.

IV. Framework Architecture

SmartBugs is written *Python3* and relies on *docker images* to execute the tools. SmartBugs requires Docker and Python3 with the modules *PyYAML*, *solidity_parser*, and *docker*. SmartBugs works with docker images available on the Docker Hub or locally. In order to be able to add a tool to SmartBugs, a docker image of the tool must be available.

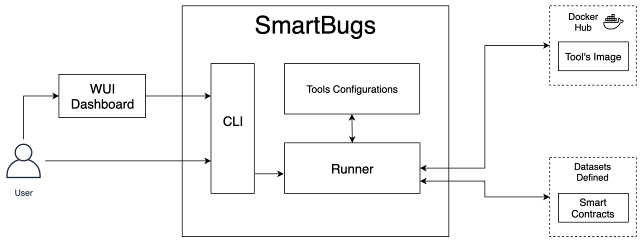

Figure 1: SmartBugs Architecture

SmartBugs is composed of five main parts plus a web-based user interface, that is integrated to interact with SmartBugs, as seen in Figure 1.

A. Methodology for adding tools

Addition of tools in SmartBugs is designed to be simple and practical, allowing the user to control the execution of the tools according to their needs. 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, and, optionally, the description of the tool and the location of the output of results. Once a docker image providing the tool is available, adding the tool to SmartBugs consists of adding a new configuration (an *.YAML* file) file.

B. Methodology for filtering bugs

SmartBugs supports the definition of *named datasets*, which is intended to represent subsets of contracts that share a common property. For example, a named dataset already provided by default is *reentrancy*: it corresponds to contracts that are identified as being vulnerable to reentrancy attacks. Named datasets can be specified in a configuration file (*config/dataset/dataset.yaml*). The default configuration file provides the named sets shown in Table IV. To add a custom named dataset, the user simply has to alter the configuration file by adding a name and the correspondent list of paths. The path can be a directory, a file, or a list of both. An example is shown below.

C. CLI & WUI

SmartBugs provides a command line interface that allows users to query the datasets defined and run different analysis tools on sets of contracts. The user can also get information about the tools, if provided, skip an execution that already has results, specify the number of processes to use during the analysis (by default 1) and list the named datasets and tools available. SmartBugs also offers a Web-based UI (WUI) built on top of SmartBugs itself. This dashboard provides the user easy access to the list of tools, named datasets available and the vulnerabilities detected by each tool available mapped to a category of DASP 10. The user has three options to analyse a smart contract: The user can paste or write a smart contract in a text box; The user can import a smart contract by uploading it to SmartBugs; The user can run tools on defined datasets. After the execution of the analysis the dashboard will present a graph with the number of security issues found by each tool, and for each tool present the issues found.

V. Analysis of 47,587 Smart Contracts

We used SmartBugs to execute the 7 automated analysis tools on two datasets. The first dataset is SB$^{\text{CURATED}}$, introduced in the previous chapter. At the time of the execution of this study SB$^{\text{CURATED}}$ was composed of 69 Solidity smart contracts with 115 manually annotated vulnerabilities (as shown in Table V), not 143 smart contracts as presented in Section III-B. All mentions to SB$^{\text{CURATED}}$ throughout this section are referring to this version. The second dataset, coined SB$^{\text{WILD}}$, contains all available smart contracts in the Ethereum Blockchain that have Solidity source code available on Etherscan[6] (a total of 47,518 contracts). We have executed 7 state-of-the-art automated static analysis tools on the two datasets and analysed 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 approximately 330 days and 15 hours to complete 333,109 analyses.

All the logs and the raw results of the analysis are available in GitHub as a repository of SmartBugs [7].

A. 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. If the time budget is spent, we stop the execution and collect the partial results of the execution. During the execution of our experiment, no tool faced timeouts.

B. SB$^{\text{CURATED}}$*: Analysis of 69 Smart Contracts*

Our goal is to compute the ability of the 7 tools in detecting the vulnerabilities present in . We executed the 7 tools on the 69 contracts, then we extracted all the vulnerabilities that

[6]Etherscan: https://etherscan.io
[7]Analysis results: https://github.com/smartbugs/smartbugs-results

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

Table V: Vulnerabilities identified per category by each tool.

Category	HoneyBadger	Mythril	Osiris	Oyente	Securify	Slither	SmartCheck	Total
Access Control	0	24	0	0	6	20	3	53
Arithmetic	0	92	62	69	0	0	23	246
Denial of Service	0	0	27	11	0	2	19	59
Front Running	0	21	0	0	55	0	0	76
Reentrancy	0	16	5	5	32	15	7	80
Time Manipulation	0	0	4	5	0	5	2	16
Unchecked Low Level Calls	0	30	0	0	21	13	14	78
Other	5	32	0	0	0	28	8	73
Total	5	215	98	90	114	83	76	681

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

were detected by the tools into a JSON file and we mapped the detected vulnerabilities by the tools to a category of vulnerabilities (see Table IV). Finally, we were able to identify which vulnerabilities the tools detect. Unfortunately, we found out that none of the 7 tools were able to detect vulnerabilities of the categories *Bad Randomness* and *Short Addresses*.

The results of the analysis of SB$^{\text{CURATED}}$ are presented in Table V and Table VI. 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 V represents a vulnerability category, and each cell presents the number of vulnerabilities where the tool detects a vulnerability of this category. 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 *Other*. 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 VI also shows that the tools offer distinct accuracies. Indeed, the tool *Mythril* has the best accuracy among the 7 tools. *Mythril* detects 27% of all the vulnerabilities when the average of all tools is 12%. Moreover, *Mythril*, *Slither*, and *SmartCheck* are the tools that detect the largest number of different categories (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*, since it detects 42 (37%) of all the vulnerabilities. This combination offers a good balance between performance and execution cost.

We now consider all the vulnerability detections and not only the ones that have been tagged in SB$^{\text{CURATED}}$. Table VI presents the total 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 V. Unsurprisingly, the more accurate a tool is in detecting known vulnerabilities, the more accurate it is at detecting unknown vulnerabilities.

C. SB$^{\text{WILD}}$*: Vulnerabilities in Production Smart Contracts*

The analysis of the contracts in SB$^{\text{WILD}}$ followed the same methodology as in the previous analysis of SB$^{\text{CURATED}}$, however, for SB$^{\text{WILD}}$, we do not have an oracle to identify the vulnerabilities.

Table VII presents the results of executing the 7 tools on the 47,518 contracts. It shows that the 7 tools are able to detect eight different categories of vulnerabilities. In total, 44.549 contracts (93%) have at least one vulnerability detected by one of the 7 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.

Since we observed a potentially large number of false positives, we analysed 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. *Arithmetic* the category with the highest consensus between four and more tools: 937 contracts are flagged as having an *Arithmetic* vulnerability

Category	HoneyBadger	Mythril	Osiris	Oyente	Securify	Slither	SmartCheck	Total
Access Control	0 0%	1,076 2%	0 0%	2 0%	614 1%	2,356 4%	384 0%	3,758 7%
Arithmetic	1 0%	18,515 39%	13,922 29%	34,306 72%	0 0%	0 0%	7,430 15%	37,590 79%
Denial of Service	0 0%	0 0%	485 1%	880 1%	0 0%	2,555 5%	11,621 24%	12,419 26%
Front Running	0 0%	2,015 4%	0 0%	0 0%	7,217 15%	0 0%	0 0%	8,161 17%
Reentrancy	19 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%	1,470 3%	1,452 3%	0 0%	1,988 4%	68 0%	4,005 8%
Unchecked Low Calls	0 0%	443 0%	0 0%	0 0%	592 1%	12,199 25%	2,867 6%	14,655 30%
Other	26 0%	11,126 23%	0 0%	0 0%	561 1%	9,133 19%	14,113 29%	28,091 59%
Total	46 0%	22,994 48%	14,665 30%	34,764 73%	8,781 18%	22,269 46%	24,906 52%	44,549 93%

Table VII: Vulnerabilities identified per category by each tool.

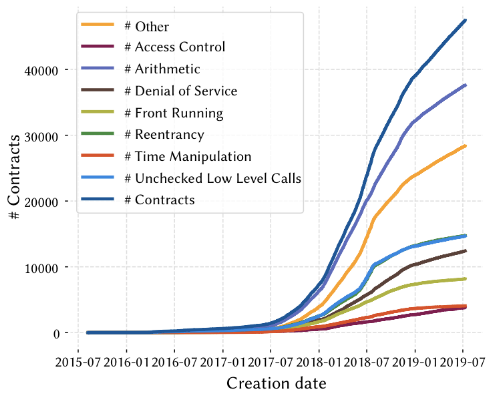

Figure 2: Evolution of number of vulnerabilities over time.

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. *Unchecked Low Level Calls* is flagged by four tools or more in 52 contracts, *Denial Of Service* and *Other* in 50 and 41, respectively. These results suggest that combining several of these tools may yield more accurate results, with less 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. So, consensus with *HoneyBadger* suggests the presence of false positives. We found that 15 contracts identified by *HoneyBadger* with vulnerabilities of type *Reentrancy* have been detected by three other tools as *Reentrancy* vulnerable.

We also analysed the evolution of the vulnerabilities over time. Figure 2 presents the evolution of the number of vulnerabilities by category. It firstly shows 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 *Other*. These two categories follow the curve of the total number of contracts. 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 indicates that there is a strong correlation between vulnerabilities in these two categories.

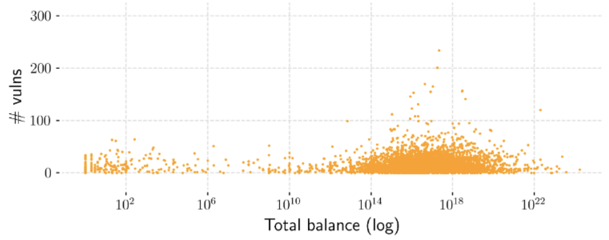

Figure 3: Correlation between the number of vulnerabilities and balance in Wei (one Ether is 10^{18} Wei).)

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. Per category, we have not observed any significant differences worth reporting.

D. Execution Time of the Analysis Tools

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.

Table VIII presents the average and total times used by each tool. In the table, we can observe two different groups of execution time: the tools that take a few seconds to execute and the tools that take a few minutes. *Oyente*, *Osiris*, *Slither*, *SmartCheck* are much faster tools that take between 5 and 34 seconds on average to analyse a smart contract. *HoneyBadger*, *Mythril*, and *Securify* are slower and take between 1m24s and 6m37s to execute. 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 really fast since they only parse the AST of the contract to identify vulnerabilities and bad practices.

Table VIII: Metrics of the tools

#	Tools	Average	Total
1	Honeybadger	0:01:38	23 days, 13:40:00
2	Mythril	0:01:24	46 days, 07:46:55
3	Osiris	0:00:34	18 days, 10:19:01
4	Oyente	0:00:30	16 days, 04:50:11
5	Securify	0:06:37	217 days, 22:46:26
6	Slither	0:00:05	2 days, 15:09:36
7	SmartCheck	0:00:10	5 days, 12:33:14
	Total	0:01:40	330 days and 15 hours

Securify and *Mythril* analyse 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. The average execution time does not reflect the complete picture of the performance of a tool. *Mythril*, for example, which has the best accuracy according to our evaluation, takes on average of 1m24s to analyse a contract. It is faster than *Securify* that only has an accuracy of 9% compared to the 27% of *Mythril*.

E. Precision of the Static Analysis Tools

For each tool output and contract analysed in Section V-B, we manually verified the issues identified by each tool on each contract and labeled them as:

- True Positive (TP): Number of true positives, i.e. pieces of code which are indeed vulnerable and the vulnerability is identified by the tool;
- False Positive (FP): Number of false positives, i.e. a vulnerability identified where it does not exist;
- False Negative (FN): Number of false negatives, i.e. the tool failed to identify a vulnerability.

Since the notions above defined do not apply equally to HoneyBadger, as it is a special case where a true positive represents that the vulnerability does not exist (it detects honeypots) and their sample of positives is not relevant (HoneyBadger had a total of five positives), we decided to not include it in this discussion to avoid confusion.

Applying the defined notions we can introduce other useful measures to assess the performance of the static analysis tools:

- False Negative Rate (FNR): FNR = FN / (FN + TP)
- False Discovery Rate (FDR): FDR = FP / (TP + FP)
- Sensitivity = 1 - FNR
- Precision = 1 - FDR

After carefully analysing all contracts for vulnerabilities, we identified a total of 167 vulnerabilities in the 69 contracts of SB$^{\text{CURATED}}$. Table IX reflects all the metrics of each tool regarding the execution of the analysis discussed in Section V-B. All informational outputs were not considered in the collection of metrics. The same methodology followed in the analysis presented in the previous section was followed.

Table IX: Metrics of the tools

	Mythril	Osiris	Oyente	Securify	Slither	SmartCheck
TP	88	59	46	31	61	49
FP	78	39	44	35	22	27
FN	79	108	121	136	106	118
FDR	46,4%	39,8%	48,9%	53,0%	26,5%	35,5%
FNR	47,3%	64,7%	72,5%	81,4%	63,5%	70,7%
Precision	53,6%	60,2%	51,1%	47%	73,5%	64,5%
Sensitivity	52,7%	35,3%	27,5%	18,6%	36,5%	29,3%

It is possible to verify that the amount of positives (TP + FP) of Mythril and Securify are not in line with the ones presented in Table VI. The reason is that Mythril and Securify usually outputs the same vulnerability more than once, making the amount of positives in Table VI bigger, as the analysis discussed in Section V-B was automated and counts all the execution output. Table IX does not take in account repeated outputs, as it resulted from a manual analysis.

F. Results

Mythril shows the best sensitivity, having the most number of true positives. As previously indicated by Table V, it is the one that flags more positives. It also seems to have the most number of false positives. Slither is the tool that presents best precision (73,5%), followed by SmartCheck.

VI. SmartCheck Extension

The analysis presented in the previous Section showed us that there is room for improvement for static analysis to detect more vulnerabilities. Bad Randomness was one of the categories that all of the tools failed to detect. In this section, we propose to extended the tool SmartCheck [27] to enable the detection of vulnerabilities related to Bad Randomness and improve detection of Time Manipulation and Access Control.

SmartCheck runs lexical and syntactical analysis on Solidity source code. It uses a custom Solidity grammar to generate a XML parse tree as an intermediate representation (IR). SmartCheck detects vulnerability patterns by using XPath patterns on the IR. Our approach to improve SmartCheck vulnerabilities detection is to investigate new rules and add them, in the form of XPath patterns. One of the problems of the SmartCheck approach, using XPath patterns, is that more complex rules can not be precisely described using XPath. So, when more complex cases are needed to be put in form of XPath patterns they can easily lead to false positives.

We propose to add three more rules to SmartCheck to improve an already established rule. Our first approach, based on the analysis presented in Chapter V, was to add a rule to detect Bad Randomness issues called *SOLIDITY_BAD_RANDOMNESS*. To do that, we created a XPath pattern to detect the use of environment variables such as: *block.number*, *block.coinbase*, *block.difficulty*, *block.gaslimit*, *blockhash* and *block.blockhash*. We followed a similar approach to update the pattern *SOLIDITY_EXACT_TIME*, already included in SmartCheck, we modified the pattern to look for expressions that contains *block.timestamp* or *now*, not only when used in comparisons as previously defined. This rules are straightforward and its goal is to simply detect the use of the referred environment variables and to flag their use, acting as a warning.

Regarding Access Control, SmartCheck only detects *tx.origin* issues. We added a pattern to search for suicides (*selfdestruct*) and ownership transfers where the function misses proper protection. To do that we constructed two rules patterns inside a single rule named *SOLIDITY_UNPROTECTED*. To detect unprotected issues we created a pattern to look for all functions defined, excluding constructors, that do not have standard *modifiers* defined, as *onlyOwner*, or *require* statements protecting a value assignment to a variable defined as *owner* or *selfdestruct* calls.

A. Results

We analysed SB$^{\text{CURATED}}$ with SmartCheck Extended, using the same version used in the study presented in Chapter **??**, to set a fair ground of comparison. Table X compares the results of SmartCheck, discussed in Section V-B, and the results obtained from executing SmartCheck Extended in the same dataset, SB$^{\text{CURATED}}$. We can observe that our extension is capable of detecting a total of 15 more issues, with regard of the annotated vulnerabilities in SB$^{\text{CURATED}}$, more than doubling the capability of detection of SmartCheck without our extension. With our proposed extension we can detect 24% of the vulnerabilities annotated in SB$^{\text{CURATED}}$, instead of the previous 11%.

Table X: Vulnerabilities identified per category by SmartCheck and SmartCheck Extended in SB$^{\text{CURATED}}$.

Category	SmartCheck	SmartCheck Extended
Access Control	2/19 11%	4/19 21%
Arithmetic	1/22 5%	1/22 5%
Bad Randomness	0/31 5%	10/31 32%
Denial of Service	0/7 0%	0/7 0%
Front Running	0/7 0%	0/7 0%
Reentrancy	5/8 62%	5/8 62%
Short Addresses	0/1 0%	0/1 0%
Time Manipulation	1/5 20%	4/ 5 80%
Unchecked Low Level Calls	4/12 33%	4/12 33%
Other	0/3 0%	0/3 0%
Total	13/115 11%	. 28/115 24%

When taking in consideration the precision metrics presented in Section V-E, where all 167 vulnerabilities present in SB$^{\text{CURATED}}$ are taken into account, our solution improves 15,1% in sensitivity and 5,1% in precision, as seen in Table XI.

Table XI: SmartCheck and SmartCheck Extended metrics.

	SmartCheck	SmartCheck Extended
TP	49	74
FP	27	32
FN	118	93
FDR	35,5%	30,2%
FNR	70,7%	55,6%
Precision	64,5%	69,6%
Sensitivity	29,3%	44,4%

Table XII: Positives of SmartCheck Extended.

	A. Control	B. Randomness	T. Manipulation
TP	2	10	13
FP	5	-	-
Warnings	-	37	4

Table XII divides all the vulnerabilities detected by our extension in TP, FP or warnings. True positives map the positives flagged from the 167 vulnerabilities taken in consideration in Section V-E. False positives are the number of issues flagged that are not true. Warnings are the number of issues detected that are not mapped in the 167 vulnerabilities, but also are not false positives. As previous stated, *SOLIDITY_BAD_RANDOMNESS* and *SOLIDITY_EXACT_TIME* are meant to act as warnings and simply flag the use of sensitive environment variables.

SmartCheck Extended is available on GitHub[8], as a fork of the original SmartCheck. It is also included in SmartBugs and ready to executed.

VII. Conclusion

As smart contracts use grows, more vulnerabilities are found and their impact becomes bigger. With our work we contributed to improve the research on static analysis tools and introduced new tools to facilitate research and automated analysis. We enumerated the state-of-the-art static analysis tools available to verify smart contracts. We introduced SmartBugs, an important contribution to automate and ease the execution of smart contracts analysers. We also introduced SB$^{\text{CURATED}}$, a dataset of 143 annotated vulnerable Solidity smart contracts with a defined vulnerability taxonomy that can serve as a reference dataset to the research community.

Using SmartBugs, we were able to organize and easily execute 7 state-of-the-art tools to perform an empirical review on 47,587 Ethereum smart contracts. We discussed the results and got important remarks and insights. We were able to detect that *Bad Randomness* is a category of vulnerabilities overlooked by the 7 tools reviewed, and thus we also contributed to the field by extending SmartCheck to detect *Bad Randomness* vulnerabilities and improved it by 15,1% in sensitivity and 5,1% regarding the non-extended version.

With our work we accomplished the objectives proposed and all the topics mentioned in the research questions were addressed, thus we are now able to provide a direct answer to all the research questions:

Answer to RQ1. In our research we listed 35 state-of-the-art tools currently available to analyse Ethereum smart contracts. Further more we listed all their URLs and identified the research paper if existent, we also reviewed 7 state-of-the-art tools.

Answer to RQ2. In Table IX we established the precision and sensitivity of six state-of-the-art tools. Slither showed to be most precise and Mythril seems to have the most sensitivity.

Answer to RQ3. In Table V are defined vulnerabilities identified per category by each tool. It seems that the categories *Other*, *Reentrancy*, *Arithmetic* and *Unchecked Low Level Calls* are the most detected. Table VII also reiterates it. With this data we can also say that they seem to be the most common type of vulnerabilities.

Answer to RQ4. The seven tools identify vulnerabilities in 93% of the contracts (in 44,589 contracts), which suggests a high number of false positives.

[8]SmartCheck Extended: https://github.com/pedrocrvz/smartcheck

Answer to RQ5. On average, the tools take 1m40s to analyze one contract. Slither is the fastest tool and takes on average only 5 seconds to analyze a contract. We have not observed a correlation between accuracy and execution time.

Answer to RQ6. By presenting an extension of SmartCheck in Sectiom VI we already contributed to improve a state-of-the-art analysis tool.

REFERENCES

[1] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," 2019, https://arxiv.org/pdf/1910.10601.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *https://bitcoin.org/bitcoin.pdf*, 2008.

[3] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[4] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*. New York, NY, USA: ACM, 2018, pp. 653–663.

[5] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. New York, NY, USA: ACM, 2016, pp. 254–269.

[6] M. di Angelo and G. Salzer, "A survey of tools for analyzing ethereum smart contracts," in *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*. Newark, CA, USA, USA: IEEE, April 2019, pp. 69–78.

[7] S. Azzopardi, J. Ellul, and G. J. Pace, "Monitoring smart contracts: Contractlarva and open challenges beyond," in *Runtime Verification*, C. Colombo and M. Leucker, Eds. Cham: Springer International Publishing, 2018, pp. 113–137.

[8] R. Norvill, B. B. F. Pontiveros, R. State, and A. Cullen, "Visual emulation for ethereum's virtual machine," in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. Taipei, Taiwan: IEEE, 2018, pp. 1–4.

[9] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey, "Erays: Reverse engineering ethereum's opaque smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1371–1385. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/zhou

[10] H. Liu, C. Liu, W. Zhao, Y. Jiang, and J. Sun, "S-gram: towards semantic-aware security auditing for ethereum smart contracts," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2018, pp. 814–819.

[11] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *Principles of Security and Trust*, L. Bauer and R. Küsters, Eds. Cham: Springer International Publishing, 2018, pp. 243–269.

[12] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "Ethir: A framework for high-level analysis of ethereum bytecode," in *Automated Technology for Verification and Analysis*, S. K. Lahiri and C. Wang, Eds. Cham: Springer International Publishing, 2018, pp. 513–520.

[13] A. Mavridou and A. Laszka, "Tool demonstration: Fsolidm for designing secure ethereum smart contracts," in *Principles of Security and Trust*, L. Bauer and R. Küsters, Eds. Cham: Springer International Publishing, 2018, pp. 270–277.

[14] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Klagenfurt, Austria: IEEE, 2017, pp. 442–446.

[15] C. F. Torres, M. Steichen, and R. State, "The art of the scam: Demystifying honeypots in ethereum smart contracts," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1591–1607. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/ferreira

[16] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu *et al.*, "Kevm: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. Oxford, UK: IEEE, 2018, pp. 204–217.

[17] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 116, 2018.

[18] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," 2019.

[19] B. Mueller, "Smashing ethereum smart contracts for fun and real profit," in *9th Annual HITB Security Conference (HITBSecConf)*. Amsterdam, Netherlands: HITB, 2018.

[20] C. F. Torres, J. Schütte *et al.*, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*. New York, NY, USA: ACM, 2018, pp. 664–676.

[21] M. Suiche, "Porosity: A decompiler for blockchain-based smart contracts bytecode," *DEF con*, vol. 25, p. 11, 2017.

[22] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. New York, NY, USA: ACM, 2018, pp. 65–68.

[23] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. Paris, France: IEEE, Feb 2018, pp. 1–5.

[24] J. Chang, B. Gao, H. Xiao, J. Sun, and Z. Yang, "scompile: Critical path identification and analysis for smart contracts," 2018.

[25] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2018, pp. 67–82.

[26] J. Feist, G. Greico, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proceedings of the 2Nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, ser. WETSEB '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 8–15. [Online]. Available: https://doi.org/10.1109/WETSEB.2019.00008

[27] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. Gothenburg, Sweden, Sweden: IEEE, 2018, pp. 9–16.

[28] P. Hegedus, "Towards analyzing the complexity landscape of solidity based ethereum smart contracts," *Technologies*, vol. 7, no. 1, p. 6, 2019.

[29] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1317–1333. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/krupp

[30] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," 2018.

[31] S. K. Lahiri, S. Chen, Y. Wang, and I. Dillig, "Formal specification and verification of smart contracts for azure blockchain," 2018.

[32] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "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*. San Diego, California, USA: NDSS, 2018, pp. 1–15.

[33] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *Principles of Security and Trust*, L. Bauer and R. Küsters, Eds. Cham: Springer International Publishing, 2018, pp. 243–269.

[34] D. Perez and B. Livshits, "Smart contract vulnerabilities: Does anyone care?" 2019.

[35] NCCGroup, "Decentralized application security project (or dasp) top 10," https://dasp.co, 2018.