2018 IEEE Confs on Internet of Things, Green Computing and Communications, Cyber, Physical and Social Computing, Smart Data, Blockchain, Computer and Information Technology, Congress on Cybermatics

Security Vulnerabilities in Ethereum Smart Contracts

Ardit Dika and Mariusz Nowostawski
Norwegian University of Science and Technology | NTNU
Information Security and Computer Science Departments, Gjovik, Norway
e-mail: dika.ardit@gmail.com, mariusz.nowostawski@ntnu.no

***Abstract*—**

Ethereum provides an open, global computing platform, that allows the exchange of value, automated and enforced workflows, and the development of general purpose applications and libraries. Smart contracts present a foundation for the computational capabilities of the Ethereum network. Motivated by the known security breaches and recurring financial losses due to smart contracts vulnerabilities, we review the field of security of smart contract programming and provide a comprehensive taxonomy of all known security issues. We achieve that by a thorough review of known vulnerabilities. In this work we also review the security code analysis tools used to identify known vulnerabilities. We conduct the investigation of security code analysis tools on Ethereum by assessing their effectiveness and accuracy on known issues on a representative sample of vulnerable contracts. We have used 21 clean, and 24 vulnerable contracts and four security tools: Oyente, Securify, Remix, and SmartCheck, to assess the quality of contemporary security analysis tools specific to Ethereum. The results indicate that there are overall inconsistencies between the tools in respect to different security properties. SmartCheck outperformed the other tools in terms of effectiveness, whereas Oyente performed the best in terms of accuracy. Furthermore, based on the limitations we identified, we propose improvements within the user interfaces, interpretation of results, and, most importantly, an enhanced list for vulnerability checks.

1. Introduction

Blockchain technology and cryptocurrencies have experienced a steady increase of attention from academia and the industry alike [1]. Blockchain technology represents a fully distributed public ledger and a peer-to-peer platform which makes use of cryptography to securely host applications, transfer digital currencies, messages, and store data [2].

One of the most popular blockchain platforms as of March 2018, based on the current cryptocurrency market capitalization[1], is **Ethereum**. Vitalik Buterin, the main inventor of Ethereum, in one of the panel discussion explains Ethereum as a general purpose blockchain, meaning that the Ethereum network is able to accomodate algorithms expressed in a general purpose programming language. This allows developers to build a variety of applications, ranging from simple wallets to financial systems, energy trading systems or new and novel crypto currencies. Instead of building a separate blockchain for each use case or application, a variety of use cases can be done through technology known as smart contracts. Ethereum can be considered a relatively new and highly experimental platform, both because of the time when it was introduced (July 2015[2]), as well as its ability to create distributed applications with a Turing-complete programming language running in a decentralised, peer-to-peer blockchain platform. A general-purpose programming language in a blockchain-based platform creates the opportunity for implementing a wide range of decentralised applications. On the other side, it creates opportunities for abuse.

In this work, we investigate the known security vulnerabilities of smart contracts and provide an updated in-depth analysis of existing smart contract vulnerabilities. In the second part, we investigate the security code analysis tools used to identify vulnerabilities and bugs in smart contracts. To the best of our knowledge this is a unique contribution to the field as the analysis of existing tools has not been conducted before.

2. Problem Description

In addition to expressing business logic and handling different, sometimes heavy computational tasks, based on the Ether price[3], smart contracts also present a foundation for possessing expensive digital assets. This means that there are currently financial and semi-financial smart contracts which are worth thousands and millions of dollars. Due to those reasons, smart contracts and the Ethereum platform are continuously a target for adversaries and manipulators. As a result, one of the main and active research areas is within the security drawbacks of high-level programming languages used for smart contract programming. The research community proposes further research and development work in formal verification, techniques for analyzing smart contracts, and defensive programming techniques.

What Ethereum and other popular blockchain platforms have in common, is the publicly visible data. This is

1. Cryptocurrency Market Capitalisation: https://coinmarketcap.com/

2. Link: https://blog.ethereum.org/2015/07/30/ethereum-launches/

3. $705 as of the time of writing: https://ethereumprice.org/

978-1-5386-7975-3/18/$31.00 ©2018 IEEE
DOI 10.1109/Cybermatics_2018.2018.00182

Authorized licensed use limited to: NED UNIV OF ENGINEERING AND TECHNOLOGY. Downloaded on July 07,2022 at 18:57:09 UTC from IEEE Xplore. Restrictions apply.

a result of having a decentralized peer-to-peer network and distributed ledger among thousands of nodes[4]. Hence, Ethereum is referred to as *The World Computer*[5]. Regardless of the positive impact and many benefits that this approach has on Ethereum and generally in any other public distributed blockchain, it presents serious challenges from the security perspective. Implementing specific use cases of smart contracts, considering the fact that the complete source code of an application is publicly visible from anyone in the network, and making sure that the code is correctly validated and verified.

One of the key characteristics of Ethereum's platform is that once you deploy your smart contract in the blockchain, you cannot modify or alter it. This characteristic can both be seen as advantageous and disadvantageous. The advantage is that it represents a trustworthy platform where the developers cannot modify the smart contract once they have deployed it, with the sole purpose of gaining illegal profit and misleading the users. The disadvantage lies in the unusual development challenges, such as the inability to easily patch discovered vulnerabilities in already deployed contracts.

Due to the above issues, a significant number of smart contracts are considered to be vulnerable. In 2016, a symbolic execution analysis tool (Oyente) was developed by Luu et al. [3], which analysed all smart contracts in the Ethereum blockchain at that time, in order to identify potential vulnerabilities[6]. Their results state, that at that time, 45% of 19,366 smart contracts in total were vulnerable with at least one security issue [3].

Our work provides insight into the smart-contract security domain. Through a thorough research and a comprehensive experiment on security vulnerabilities and code analysis tools we propose up-to-date taxonomy of vulnerabilities, their architectural classification, in conjunction with their severity level. In addition, we conduct an experiment on several security tools to assess their accuracy, effectiveness, and consistency. This generates results, such as false positive and false negative rates and an overall discussion on how effective these tools are in analysing the smart contracts from the data collected in this study.

3. Related Work

We focus on three aspects in the area of smart contracts: security vulnerabilities, smart contract attacks/incidents, and preventive methodologies. In order to identify the current state of the art, we have conducted literature review. Some of the existing research is focused on security vulnerabilities in general [3], [4], [5], [6], [7]. Others are focused on specific vulnerabilities and smart contract challenges, such as privacy [8], [9]. There are also articles focused only on one specific vulnerability, for example, on timed commitments [10] and smart contract altering possibilities [11].

One of the most established taxonomies in this area is the one provided by Atzei et al. [4]. The levels chosen to represent the vulnerabilities are (i) focused on the programming language Solidity, (ii) specific to the underlying implementation of the EVM, and (iii) specific to the Blockchain itself. Note, *Solidity vulnerabilities* are also applicable for other high-level programming languages in Ethereum. This taxonomy seems to properly classify all vulnerabilities based on their level, since a newly discovered vulnerability falls in one of these categories. Therefore, we have used it as a basis for our own taxonomy.

Another taxonomy is provided by Alharby and Moorsel [5]. This one is based on a systematic study of current research topics related to smart contracts, and it identifies four key smart contract issues; *codifying issues, security issues, privacy issues*, and *performance issues*. By *codifying issues* they refer to the challenges that are related with the development of smart contracts. This could be generally called validation issues. The *security issues* mean bugs or vulnerabilities (verification), and *privacy issues* are related to unintentional information disclosures. Lastly, *performance issues* are related to the challenges that affect the ability of blockchain to scale. [5]

Other research articles refer to security vulnerabilities in general, without any categorisation, such as, in [3], where they discuss only severe vulnerabilities. In [7], where Buterin with the community's help created a crowd sourced list of the major bugs with smart contracts, and in [6] through a university course for smart contract programming, they exposed numerous common pitfalls and vulnerabilities.

In addition to the above, more general research on vulnerabilities, there is research on specific vulnerability, for example *privacy preserving* issues. It represents a category of development challenges to keep critical functions secret, apply correctly cryptographic protocols, and avoid disclosing data that should not have been public in the first place. A research on 'replacing paper contracts with Ethereum smart contracts' finds out what kind of criteria Ethereum needs to fulfil to be properly applied on replacing paper contracts [8]. They conclude that due to a large privacy setback it is not *yet* recommended to replace legally-enforceable agreements with smart contract applications [8]. This is as a result of the private information these papers (agreements) hold and the damage that could be done if they become public or if the blockchain does not work as intended on preserving privacy. According to [5], *lack of transactional privacy* and *lack of data feeds privacy* are two issues correlated with the privacy preserving category.

A similar research on the issue of **privacy-preserving** is conducted by Kosba et al. [9]. They highlighted the significant importance of privacy in smart contract applications generally in blockchain technologies, not only in Ethereum. For a solution to this issue, they have proposed a decentralized smart contract system, Hawk, which does not store financial transactions in the blockchain and saves the developers from implementing any cryptographic function-

4. Ethereum as of May, 2017 has nearly 25,000 nodes. Link: http://www.trustnodes.com/2017/05/31/ethereum-now-three-times-nodes-bitcoin

5. Ethereum: the World Computer: https://www.youtube.com/watch?v=j23HnORQXvs

6. Vulnerabilities that Oyente is able to identify: transaction-ordering dependence, timestamp dependence, mishandled exceptions and reentrancy.

Authorized licensed use limited to: NED UNIV OF ENGINEERING AND TECHNOLOGY. Downloaded on July 07,2022 at 18:57:09 UTC from IEEE Xplore. Restrictions apply.

ality [9]. Juels et al. [12], also investigated the leakage of confidential information and theft of cryptographic keys for smart contracts used in criminal activities.

One of the most prominent vulnerabilities of Ethereum is considered to be the *timestamp dependency*. Boneh and Naor [10], introduce and construct *timed commitment schemes* which are proposed as a solution for this vulnerability. Their proposed solution could be applied when two mutually suspicious parties wish to exchange signatures on a contract.

Another issue that has been tackled in the literature is the *gas-costly pattern*, more specifically, the under-optimised smart contracts that consume more gas than necessary. A research investigation in this regard is done by Chen et al. [13], in which they identified 7 gas costly patterns and grouped them into two categories. They also developed a tool, named Gasper, focused only on identifying gas-costly patterns by analysing the smart contracts' bytecode [13]. Their results indicate that over 80% of 4240 smart contracts analysed, suffer from one of the gas-costly patterns [13].

In addition, there are specific challenges in communicating with **external services** (Oracles[7]). Zhang et al. [14] presented an authenticated data feed system called Town Crier, which enables smart contracts to consume data from outside the blockchain while preserving confidentiality with encrypted parameters.

There is a significant number of issues, some of which have known solutions. Some of the solutions proposed require for blockchain upgrades, meaning that all the nodes have to upgrade their version in order to solve a particular issue, or they are proposed as a separate platform on top of a blockchain. This makes it challenging in rolling out the actual fixes.

4. Ethereum Vulnerabilities

In this section we provide a brief explanation for each of the security vulnerabilities that has been included into the taxonomy. Some of these vulnerabilities have already been known for a while, therefore we follow well-established naming conventions. Some small self-explanatory issues with the Ethereum smart contracts are excluded from the list and the main focus of this section is (mostly) within the severe vulnerabilities.

Reentrancy is considered to be one of the most severe vulnerability. It has been first recognised by the biggest attack ever made (TheDAO hack). The reentrancy vulnerability relies on the interaction between two smart contracts, (A) and (B). If through an interaction from a contract (A) with another contract (B), (A) handing over control to contract (B) makes it possible for (B) to call back into (A) before the first initiated interaction is completed, contract (B) can effectively retrieve multiple refunds and empty the balance of contract (A). The use of *checks-effects-interactions*[8] is recommended as a solution to avoid this vulnerability.

tx.origin (transaction origin) is the identity of the user who initiated a chain of interactions between contracts. The usage of tx.origin for authorisation is discouraged, as it is easy for attacker to spoof that value in a contract. tx.origin must be used with extra care not to allow an attacker obtaining leveraged privileges in the contract.

Callstack depth exception. It is possible to makes an external call to fail because it exceeds the maximum call stack of 1024 [15]. As a result, the call will fail, and if the exception is not properly handled by the contract, the attacker can force the contract to produce an output which suits them.

Timestamp dependence presents a common vulnerability favouring a malicious miner. If a contract is using it for a critical check, the miner can manipulate the timestamp for a few seconds by changing the output to be in its favour [16]. However, this vulnerability is severe only if used in critical components of a contract and requires miners to have sufficient computing facilities.

Transaction-ordering dependence refers to the idea that the user can never be sure of the order of transactions. For example consider a smart contract which offers a reward for solving a puzzle. Once a user solves the puzzle and submits the transaction, at the same moment the smart contract owner can reduce (or completely remove) the reward. There is a probability that the transaction that reduces or removes the reward is processed first. In this case, the owner gets an answer for the puzzle, and the solver (user) does not get the reward.

The use of **external calls** is considered to be by default risky [16], because adversaries can execute malicious code in that external contract. Therefore, it is recommended to possibly avoid external calls ("calls to the unknown") in general or treat those calls as potentially risky and take precautions, such as, use *send* instead of *call_value()*, favor *pull* over *push* for external calls, and handle errors (check the return value) [16].

Unchecked-send bug is part of the **exception disorders** or **mishandled exceptions**. This class of vulnerabilities is also referred to as "*send* instead of *transfer*. '*Transfer*' automatically checks for the return value, whereas using '*send*' you have to manually check for the return value, and throw an exception if the send fails. Not doing so, can lead to an attacker executing malicious code into the contract and draining the balance. Overall, the consequences are similar to the **reentrancy** and **call to the unknown** vulnerability.

DoS is explained by SmartCheck[9] as a situation in which conditional statement *(if , for , while)* depends on an external call: the callee may permanently fail *(throw or revert)*, preventing the caller from completing the execution [17]. An attacker can cause inconvenience by supplying the contract with data that is expensive to process, thereby preventing

7. Oracle: A reliable connection between Web APIs and smart contracts, since smart contracts cannot fetch external data on their own.

8. First subtract the value from the contracts' balance then send the Ether, and check for the return value.

9. SmartCheck: https://smartcontracts.smartdec.net/

Authorized licensed use limited to: NED UNIV OF ENGINEERING AND TECHNOLOGY. Downloaded on July 07,2022 at 18:57:09 UTC from IEEE Xplore. Restrictions apply.

others to interact with it. This vulnerability is closely related to the **external calls** vulnerability and to prevent this form of attack from happening, we need to handle properly any *throw* exceptions from external calls, and also, avoid looping behaviour.

Blockhash usage similarly to the block timestamp, it is not recommended to be used on crucial components, for the same reason as with the timestamp dependency, because the miners, to some degree, can manipulate it and change the output to their favour. This is particularly pronounced when blockhash is used as a source of randomness.

Gasless send makes a transaction to fail if not enough gas is provided for a specific call. The maximum gas limit on the network can vary over time based on the transaction fees[10]. It is important to throw an exception if a failure based on the gas consumption happens. Also, it is important to develop functions that do not require too much gas, not only for the purpose of failing, but also for the sole purpose of mitigating the costs of executing the contract.

Other vulnerabilities include; **immutable bugs** (e.g. wrong constructor name) which refer to a bug or a code mistake which cannot be altered after deployment/discovery, **the use of untrustworthy data feeds**, **failure to keep secrets** or in other words failure to apply cryptography and as a result expose crucial functions or values, the **challenge to generate randomness**, **style guide violation**, and others.

Existing defensive programming as well as traditional secure programming techniques apply to smart contract programming. Note however, that many of vulnerabilities are unique to smart contracts. Some of it has to do with the novel approach of building smart contract applications in a public blockchain. Since, so far, developers are used to a more traditional way of developing software. Traditionally, developers do not have to worry about many of these issues, because the traditional systems provide certain guarantees that a public blockchain cannot.

5. Preventive Methodologies

ZeppelinOS is an operating system for smart contract applications developed by Zeppelin Solutions [18]. As referred to by Zeppelin Solutions, ZeppelinOS is "an open-source, distributed platform of tools and services on top of the EVM to develop and manage smart contract applications securely". Their system is composed of four components, *kernel, scheduler, marketplace*, and *off-chain tools*. In other words, Zeppelin introduces a novel approach in developing smart contracts by using already developed and secure smart contracts (i.e. libraries). The off-chain component provides numerous tools like debugging, testing, deployment and monitoring. Based on their team, these tools will enhance the development process (better, easier, robust), and generally will help to provide a more secure smart contract environment.

SolCover[11] provides code coverage for Solidity testing. Relying on code coverage, SolCover measures and describes the degree of overall testing in a smart contract. Even though, it does not serve as a mechanism to identify specific vulnerabilities, it could be argued that it creates a more secure environment with the philosophy that more tests provide improved security metrics.

HackThisContract[12] is a crowdsourcing experimental website that encourages developers to test smart contracts before deployment by uploading it on their website. Other developers, with their own techniques, will try and exploit possible vulnerabilities. Additionally, they provide a list of vulnerable smart contract examples which the developers should not follow. Overall, with the sole purpose of deploying secure smart contracts and mitigate severe issues in a pre-deployment phase.

Security audits are considered to be the most effective way of identifying vulnerabilities in a pre-deployment phase. Experienced blockchain developers and specialised teams carefully investigate the smart contract manually and automatically to identify vulnerabilities. Despite the fact that it might be the most secure method for preventing deployment of vulnerable smart contracts, it is not popular because of the high costs and time it takes to conduct them[13]. Currently there are many firms that do smart contract security audits: Zeppelin, Solidified[14], SmartDec[15], and DejaVu[16].

Other preventive methodologies include staying up-to-date with Ethereum upgrades and especially with the attacks that happen over time, since they may discover a new vulnerability. Also, it is of a vital importance to follow a list of recommendations for secure smart contracts once you start developing, such as the extensive list by ConsenSys [19].

Oyente. Oyente is known to be the first and most popular security analysis tool. It was developed by Luu et al. [3] and is one of the few tools presented in a major security conference, Ethereum Devcon[17]. Oyente leverages symbolic execution to find potential security vulnerabilities, including here **transaction-ordering dependence, timestamp dependence, mishandled exceptions** and **reentrancy**. The tool can analyze both Solidity, and the bytecode of a smart contract. In its early stage it could have been used only through a command line interface. Currently, it provides a more user-friendly web-based interface. It is worth mentioning that it is the only tool that describes its verification method to eliminate false positives [3].

Securify. Securify[18] is a web-based security analysis tool and, according to their website, it is the first security analysis tool that provides *automation* (to enable everyone to verify smart contracts), *guarantees* (for finding specific vulnerabilities), and *extensibility* (to capture any newly dis-

10. Link:http://www.kingoftheether.com/contract-safety-checklist.html
11. Link: https://github.com/sc-forks/solidity-coverage
12. Link: http://hackthiscontract.io/
13. Based on a community discussion in Reddit, a smart contract security audit costs between $20k-$60k. Link: https://www.reddit.com/r/ethdev/comments/6pdgvd/how_much_does_a_smart_contract_audit_cost/
14. Link: https://solidified.io/
15. Link: https://smartcontracts.smartdec.net/
16. Link: http://www.dejavusecurity.com/services/
17. Link: https://www.youtube.com/watch?v=bCvh6ED-cj0
18. Link: https://securify.ch/

Authorized licensed use limited to: NED UNIV OF ENGINEERING AND TECHNOLOGY. Downloaded on July 07,2022 at 18:57:09 UTC from IEEE Xplore. Restrictions apply.

covered vulnerability). Securify uses formal verification but also relies on static analysis checks. The security issues that it covers are: *transaction reordering, recursive calls, insecure coding patterns, unexpected ether flows*, and *use of untrusted input*. However, the recursive calls, unexpected ether flows, and part of the insecure coding patterns checks are locked (require full access)[19].

Remix. Remix[20] is a web-based IDE that facilitates writing Solidity smart contracts, deploying and running them. A debugger and a testing environment (test-blockchain network) are integrated. Additionally, it serves as a security tool by analyzing the Solidity code only, to reduce coding mistakes and identify potential vulnerable coding patterns. Some of the vulnerabilities that it identifies are: *tx.origin usage, timestamp dependence, blockhash usage, gas costly patterns, check effects (reentrancy)*[21]. Remix security analysis rely on formal verification (deductive program verification and theorem provers).

SmartCheck. SmartCheck[22] is also a web-based security code analysis tool provided by SmartDec team[23]. SmartDec is a company focused on security audits, analysis tools and web development. Recently (November, 2017), they released a beta version of their security tool, SmartCheck. It automatically checks for vulnerabilities and bad coding practises. In addition to that, it highlights the vulnerability (e.g. line of code), gives an explanation of the vulnerability, and a possible solution to avoid a particular security issue. Their analysis uses Solidity code and it is not stated which specific methodology they use to identify the vulnerabilities (e.g. symbolic execution, formal verification, etc.). Each vulnerability discovered is shown in correlation with its severity level. Some of the severe vulnerabilities they identify are: *DoS by external contract, gas costly patterns, locked money, reentrancy, timestamp dependency, tx.origin usage, and unchecked external call*. Additionally, SmartCheck identifies many other vulnerabilities with low severity (warnings), such as, compiler version not fixed, style guide violation, and redundant functions.

F* Framework. *F**, from Microsoft Research, presents a framework for analyzing the runtime safety and the functional correctness of Ethereum smart contracts, outlined by Bhargavan et al. [20]. It relies on formal verification, by translating Solidity or bytecode into F* (a functional programming language) and then identifying potential vulnerabilities, such as, *reentrancy* and *exception disorders*.

Mythril. Mythril is a recently released experimental security analysis tool from ConsenSys[24]. Through a command line interface, it is able to analyze bytecode, and by installing solc (command line compiler) it also analyses Solidity code. So far, it is able to identify a variety of vulnerabilities, such as, *unprotected functions, reentrancy, integer overflow/underflow,* and **tx.origin usage**. Some other severe vulnerability checks are presented as work in progress, such as, timestamp dependence, transaction-ordering dependence, and information exposure[25].

Gasper. Gasper is a security tool developed by Chen et al. [13], which is not released yet. However, from their research paper, we already know that it is focused only on identifying **gas costly programming patterns** in a smart contract through a command line interface. It runs analysis only for the bytecode. Moreover, they have discovered seven gas costly patterns, and grouped them into two categories. Gasper also relies on symbolic execution to cover all reachable code-blocks by disassembling its bytecode using *disasm* (disassembler). So far, they only cover the gas costly patterns from the first category that they have discovered, the rest is work-in-progress. [13]

6. Methodology and Experiments

	Vulnerability	Severity level
Blockchain	Unpredictable state (dynamic libraries)	2
	Generating randomness	2-3
	Time constrains / Timestamp dependence	1-3
	Lack of transactional privacy	1-3
	Transaction-ordering dependence	2-3
	Untrustworthy data feeds (oracles)	3
EVM	Immutable bugs/mistakes	3
	Ether lost in transfer	3
Solidity	Gas costly patterns	1-2
	Call to the unknown	3
	Gasless send	3
	Exception disorders / Mishandled exceptions / Unchecked-send bug	3
	Type casts	2
	Reentrancy	3
	Unchecked math (Integer over- and underflow)	1-2
	Visibility / Exposed functions or secrets/ Failure to use cryptography	2-3
	'tx.origin' usage	3
	'blockhash' usage	2-3
	DoS	3
	'send' instead of 'transfer'	1-2
	Style violation	1
	Redundant fallback function	1

TABLE 1. TAXONOMY OF VULNERABILITIES

We have assessed the smart contract security tools based on their:

- **effectiveness** - check how many smart contract problems the tools were able to find from our data set
- **accuracy** - assessing the correctness of the results they produce, based on false positive and false negative rates
- **consistency** - assessed in security tools that analyse both bytecode and Solidity, and check if there is any inconsistency[27].

We collected known vulnerabilities based on literature search and online resources, and stayed up-to-date with any related web-articles or blogs and Reddit forums that were assessing smart contract security issues, between February and October 2017. Additionally, we have used group chats (Slack channels, Gitter) and e-mails to communicate with developers or users for a specific security tool.

Audited Smart Contracts. In order to assess the false positive rates we need secure/trusted and tested smart contracts which are considered to be bug-free or at least without

19. As of October, 2017
20. Link: https://remix.ethereum.org/
21. Link: https://remix.readthedocs.io/en/latest/analysis_tab.html
22. Link: http://tool.smartdec.net
23. Link: https://smartcontracts.smartdec.net/
24. Link: https://github.com/b-mueller/mythril/
25. Link: https://github.com/b-mueller/mythril/blob/master/security_checks.md
27. For example, if a tool produces some results with the bytecode of a smart contract and other with the Solidity source code of the same contract.

Authorized licensed use limited to: NED UNIV OF ENGINEERING AND TECHNOLOGY. Downloaded on July 07,2022 at 18:57:09 UTC from IEEE Xplore. Restrictions apply.

Security Tool	ReEntrancy	Timestamp dependency	TOD[26]	Mishandled exceptions	Immutable Bugs	tx.orgin usage	Gas costly patterns	Blockhash usage
Oyente	✓	✓	✓	✓	✓	X	X	X
Remix	✓	✓	X	✓	X	✓	✓	✓
F*	✓	X	X	✓	X	X	X	X
Gasper	X	X	X	X	X	X	✓	X
Securify	✓	X	✓	✓	X	✓	X	X
S. Analysis	X	X	X	✓	X	X	X	X
SmartCheck	✓	✓	✓	✓	✓	✓	✓	X
Imandra	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Mythril	✓	X	X	✓	✓	✓	X	X

TABLE 2. TOOLS/VULNERABILITIES MATRIX

any severe security vulnerability. For that, we decided to use smart contracts which were previously audited. We chose Zeppelin[28]. We have collected 28 audited smart contracts in total from Zeppelin, starting from the one audited first, up until the last one (October 23, 2017). We did a manual check for each security audit to dismiss a smart contract which had one of the following cases:

- is written in a programming language other than Solidity,
- is identified with severe vulnerabilities from Zeppelin, and not updated afterwards,
- is used for token pre-sale[29],
- very recent security audit (not updated, nor published).

After doing this data clean-up, we ended up with a total of 21 security audited smart contracts. Table 3 provides a list of the data-set for this category, where the seven highlighted in red are the discarded smart contracts which were not taken into consideration. Additionally, each one has a link to the corresponding source on Zeppelin's blog. Lastly, the source code for each smart contract is collected using EtherScan[30].

Vulnerable Smart Contracts. Vulnerable smart contracts are used to identify the false negative rates, as well as the gaps, i.e. the vulnerabilities which are not covered by the tools. We have used existing research articles, online resources, and community discussions assessing attacks and bugs in smart contracts, to assemble a list of vulnerable smart contracts [3], [7], [21], [22], [23]. Table 4 provides the list of contracts used as our data-set.

Smart contracts which are synthetic are labelled with the term "Sample" on their name. Moreover, the ones highlighted in red are not taken into consideration, because i) *Suicide* function has been called and their code is no longer available (two cases) or ii) Smart contracts that allow their owners to withdraw the contract funds, are removed because it was considered to be more of a trust issue rather than a bug or vulnerability. Therefore, out of 28 smart contracts in total, after clean-up we ended up with 24[31]. The tools which are chosen for the experiment are: **Oyente, Securify, Remix** and **SmartCheck**. All four tools have a web-based user interface.

28. More than $450 million have been raised by smart contracts that have been audited by Zeppelin.

29. They are mostly temporary smart contracts used to crowd-fund an organisation.

30. Link: https://etherscan.io/

30. TOD: Transaction-ordering dependence

31. Stricly speaking, 23, since two smart contracts have either bytecode or Solidity available, not both.

Smart Contract	Source (* = https://blog.zeppelin.solutions)
Hacker Gold (HKG)	*/ethercamps-hacker-gold-hkg-public-code-audit-b7dd3a2fe43b
ArcadeCity (ARC)	*/arcade-city-arc-token-audit-9071fa55a4e8
Golem Network	*/golem-network-token-gnt-audit-edfa4a45bc32
ProjectKudos	*/ethercamps-projectkudos-public-code-audit-179ee0c6672d
EtherCamp's DSTC	*/ethercamps-decentralized-startup-team-public-code-audit-65f4ce8f838d
SuperDAO Promissory	*/draft-superdao-promissory-token-audit-2409e0fe776c
SuperDAO ConstitutionalDNA	*/draft-superdao-promissory-token-audit-2409e0fe776c
ROSCA	*/wetrust-rosca-contract-code-audit-928a536c5dd2
Matchpool GUP	*/matchpool-gup-token-audit-852a70330f2
iEx.ec RLC	*/iex-ec-rlc-token-audit-80abd763709b
Cosmos	*/cosmos-fundraiser-audit-7543a57335a4
Blockchain Capital (BCAP)	*/blockchain-capital-token-audit-68e882d14f0
WingsDAO	*/wingsdao-token-audit-f39f800a1bc1
Moeda	*/moeda-token-audit-ac72944caa6f
Basic Attention	*/basic-attention-token-bat-audit-88bf196df64b
Storj	*/storj-token-audit-32a9af082797
Metal	*/metal-token-audit-d7e4dbf17bcf
Decentraland MANA	*/decentraland-mana-token-audit-ee56a6bca708
Tierion Pre-sale	*/tierion-presale-audit-ec14b91c3140
Serpent Compiler	*/serpent-compiler-audit-3095d1257929
Hubbi	*/hubii-token-audit-227c0adf50ea
Tierion	*/tierion-network-token-audit-163850fd1787
Kin	*/kin-token-audit-121788c06fe
Render	*/render-token-audit-2a078ba6d759
Fuel	*/fuel-token-audit-30cc02f257f5
Enigma	*/enigma-token-audit-91111e0b7f8a
Global Messaging	*/global-messaging-token-audit-865e6a821cd8
Ripio	*/ripio-token-audit-abe43b887664

TABLE 3. AUDITED SMART CONTRACTS COLLECTION

In total, 23 vulnerable and 21 audited smart contracts are analyzed with the four tools. Since each security tool identifies different vulnerabilities, not all vulnerable smart contracts were fit to be tested with all the tools. However, we decided to analyze all vulnerable smart contracts, in order to capture a general analysis on how many vulnerabilities each tool is not able to identify. This also gives us an insight within the possible future improvements of the security tools. The data analysis consists of four different assessments: effectiveness, accuracy, consistency, and overall assessment. With the exception of the *Overall assessment*, all other three assessments have a clear data analysis process and an evaluation method.

Effectiveness. Generally, the effectiveness of the tools is assessed based on the percentage of the smart contracts in total that the tools were able to analyse. The nature of the data-set consists of different type of smart contracts, including here; secure, vulnerable, old compiler versions, grand scale, small scale, and samples. Additionally, the symbolic execution methodology predominantly used to identify vulnerabilities is rather complex, since it analyses the code without any known input and also loops through the blockchain to cover all possible behaviours. Therefore,

Authorized licensed use limited to: NED UNIV OF ENGINEERING AND TECHNOLOGY. Downloaded on July 07,2022 at 18:57:09 UTC from IEEE Xplore. Restrictions apply.

Smart contract name	Vulnerability
TheDao	Re-entrancy
SimpleDao Sample compiler version 0.3.1	Re-entrancy, call to the unkown
SimpleDao Sample compiler version 0.4.2	Re-entrancy, call to the unkown
King of the Ether game (KoET)	Unchecked-send bug, Gasless send, Mishandled exception
KotET Sample compiler version 0.3.1	Gasless send
KotET Sample compiler version 0.4.2	Gasless send
GovernMental (PonziGovernmental)	Unchecked-send bug, Call-stack limit
GovernMental simplified sample 0.3.1	Immutable bugs, exception disorder, call-stack limit, unpredictable state
Rubixi	Immutable bugs, wrong constructor name
FirePonzi	Type casts (intentional scam)
Parity Multisig 1	Unintended function exposure
Parity Multisig 2 - Suicide Function called	Unintended function exposure
Parity Multisig 3- Suicide Function called	Unintended function exposure
GoodFellas	Typo (wrong constructor name)
StackyGame	Typo (wrong constructor name)
DynamicPyramid	Contract that does not refund
GreedPit	Contract that does not refund
NanoPyramid	Contract that does not refund
Tomeka	Contract that does not refund
Double3	Allows the contract owner to withdraw all the funds
TheGame	Allows the contract owner to withdraw all the funds
ProtectTheCastle	Call-stack limit, Withdraw option
RockPaperScissors (RPS)	Public moves
SmartBillions	Blockhash bug
EtherPot	Unchecked-send bug
TheRun	Timestamp dependence
OddsAndEvents Compiler 0.3.1 Sample	Keeping secrets
OddsAndEvents Compiler 0.4.2 Sample	Keeping secrets

TABLE 4. VULNERABLE SMART CONTRACTS COLLECTION

the security tools and the methodology are themselves prone to errors and failures.

Accuracy. Assessing just the effectiveness of the tools does not necessarily show us how accurate the results are. Therefore, it is crucial to assess the accuracy of the results that the tools produce. Accuracy is assessed through the false positive and false negative rates. Initially, this assessment idea came from Zhang et al. [24], in which they evaluate the anti-phishing tools with the same methodology, using 200 verified phishing URLs (in our case vulnerable contracts) and 516 legitimate URLs (in our case audited contracts), to test the performance of 10 popular phishing tools (in our case 4 popular Ethereum security code analysis tools).

First, we ran 21 audited smart contracts in each tool. Based on the results obtained and the severity level of vulnerabilities, we decided to manually analyse only five vulnerabilities[32]. Other vulnerabilities are not considered for manual analysis, either because they cannot be manually analysed (e.g. gas costly patterns), the security audit firm does not cover them, or they are vulnerabilities with low severity (e.g. useful warnings or style violations). The manual analysis is conducted as follows:

- Check the Zeppelin source of the smart contract in which a vulnerability is identified
- If the vulnerability is also identified by Zeppelin, and the smart contract owners have not modified the code for that specific vulnerability or they have suppressed it – it is removed from the false positives results.
- Additionally, a manual analysis following a list of recommendations for smart contract security [19] is conducted and the line where the vulnerability is identified is checked manually to verify if it is false positive.

The other approach in regards to accuracy is the false negative assessment. This is done through the vulnerable contracts that have at least one vulnerability. If the tools state that they are able to identify a specific vulnerability and they fail to do so, it is considered a false negative. The results obtained from this experiment have two possibilities of failure:

- **False Positive** when the tool identifies a vulnerability in an audited smart contract, and the manual inspection does not confirm it.
- **False Negative** when the security tool does not find a specific vulnerability in a vulnerable contract.

Security Tool	Method	Bytecode analysis	Solidity analysis	CLI[33]	WUI[34]
Oyente	Symbolic execution	✓	✓	✓	✓
Remix	Formal verification	X	✓	✓	✓
F* Framework	Formal verification	✓	✓	✓	X
Gasper	Symbolic execution	✓	X	N/A	N/A
Securify	Formal verification	✓	✓	X	✓
Simple Analysis[35]	Heuristics	✓	X	✓	X
SmartCheck	N/A	X	✓	X	✓
Imandra Contracts	Formal verification	N/A - paid access			
Mythril	Concolic testing (symbolic execution)	✓	✓	✓	X

TABLE 5. TAXONOMY OF TOOLS

Table 5 provides an overview of the generated taxonomy for security code analysis tools. The categorisation is based on their similarities, such as, the methodology they use (highlight) to identify security issues (symbolic execution, formal verification), which code analysis they are able to perform (bytecode, Solidity), and their user interface (CLI, WUI). As it can be seen, for some tools we have partial information, either because the tool is not released yet (Gasper), the methodology is not stated in their documentation (SmartCheck), or the tool requires paid access for additional information and usage (Imandra Contracts).

Compared to the taxonomy provided by Hildenbrandt et al. [22], which covers all Ethereum software quality tools, our taxonomy is only focused on security tools used to identify vulnerabilities/bugs in smart contracts. The security tools stated here use symbolic execution and formal verification as a methodology to identify vulnerabilities. These two methodologies, generally, are used interchangeably and in combination. Table 2 provides the generated matrix of the security tools and the vulnerabilities they cover. The total list of vulnerabilities is extensive, where for example,

32. Including here: reentrancy, timestamp dependence, transaction reordering, unchecked-send bug, tx.origin usage.

33. CLI: Command Line Interface

34. WUI: Web-based User Interface

35. A simple program analysis tool specifically used for detecting *unchecked-send bug*. Link: http://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/

Authorized licensed use limited to: NED UNIV OF ENGINEERING AND TECHNOLOGY. Downloaded on July 07,2022 at 18:57:09 UTC from IEEE Xplore. Restrictions apply.

only SmartCheck identifies 21 vulnerabilities in total, including here various warnings and low-risk vulnerabilities. Therefore, due to space limitations, only the most important vulnerabilities are taken into consideration for this matrix. Most security tools (6 out of 8) identify more than one vulnerability, and only two tools identify one vulnerability each, Gasper (gas costly patterns), and Simple analysis (unchecked-send bug). Furthermore, since Imandra requires paid access, we do not have any information on what kind of vulnerabilities it covers. To simplify the matrix, in the *mishandled exceptions* we cover: *exception disorders*, *unchecked-send bug*, and *gasless send*. Whereas, in the *immutable bugs* category we cover, *type casts* and *integer over- and underflow* as well. *Visibility* (function exposure) checks are omitted because they are covered only from SmartCheck. And since the *stack-size limit* is not a vulnerability anymore, it is eliminated from the list, even though Oyente still has that vulnerability check.

7. Conclusion

The main purpose of this article was to provide insights into the security vulnerabilities on Ethereum smart contracts and assess the overall effectiveness of popular security code analysis tools used to detect those vulnerabilities. The main motivation behind this work was to contribute to a more secure and trustworthy Ethereum environment. We have conducted a comprehensive review on peer-reviewed publications and online resources to collect available data and to propose two taxonomies. The first taxonomy, presented in Table 1, outlines already exploited vulnerabilities and classifies them based on their architectural and severity level. It serves as a list of issues that can aid developers who plan to develop smart contract applications. The second one, to is a novel taxonomy of current security tools. We have classified the tools based on the methodology they use, the user interface, and the analysis they are able to execute, which allows us to build a 'state of the art' of security tools on Ethereum. Lastly, we construct a matrix of security tools and the vulnerabilities they cover in order to identify gaps and absent vulnerability checks.

References

[1] Z. Zheng, S. Xie, H.-N. Dai, and H. Wang, "Blockchain challenges and opportunities: A survey," *Work Pap*, 2016.

[2] C. Dannen, *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*, 1st ed. Berkely, CA, USA: Apress, 2017.

[3] 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*. ACM, 2016, pp. 254–269.

[4] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International Conference on Principles of Security and Trust*. Springer, 2017, pp. 164–186.

[5] M. Alharby and A. van Moorsel, "Blockchain-based smart contracts: A systematic mapping study," *arXiv preprint arXiv:1710.06372*, 2017.

[6] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 79–94.

[7] V. Buterin, "Thinking about smart contract security - ethereum blog," https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/, June 2016, (Accessed on 11/18/2017).

[8] W. Egbertsen, G. Hardeman, M. van den Hoven, G. van der Kolk, and A. van Rijsewijk, "Replacing paper contracts with ethereum smart contracts," 2016.

[9] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 839–858.

[10] D. Boneh and M. Naor, "Timed commitments," in *Advances in Cryptology—Crypto 2000*. Springer, 2000, pp. 236–254.

[11] B. Marino and A. Juels, "Setting standards for altering and undoing smart contracts," in *International Symposium on Rules and Rule Markup Languages for the Semantic Web*. Springer, 2016, pp. 151–166.

[12] A. Juels, A. Kosba, and E. Shi, "The ring of gyges: Investigating the future of criminal smart contracts," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 283–295.

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

[14] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 270–282.

[15] Solidity, "Security considerations — solidity 0.4.19 documentation," http://solidity.readthedocs.io/en/latest/security-considerations.html, (Accessed on 11/19/2017).

[16] Ethereum-Wiki, "Safety · ethereum/wiki wiki," https://github.com/ethereum/wiki/wiki/Safety, (Accessed on 11/19/2017).

[17] SmartDec, "Smartcheck | knowledgebase | dos by external contract," https://tool.smartdec.net/knowledge/SOLIDITY_DOS_WITH_THROW, (Accessed on 12/14/2017).

[18] M. Araoz, "Introducing zeppelinos: the operating system for smart contract applications," https://blog.zeppelin.solutions/introducing-zeppelinos-the-operating-system-for-smart-contract-applications-82b042514aa8, July 2017, (Accessed on 11/20/2017).

[19] ConsenSys, "Recommendations for smart contract security in solidity - ethereum smart contract best practices," https://consensys.github.io/smart-contract-best-practices/recommendations/, (Accessed on 11/21/2017).

[20] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Beguelin, "Formal verification of smart contracts," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security-PLAS'16*, 2016, pp. 91–96.

[21] M. Bartoletti, S. Carta, T. Cimoli, and R. Saia, "Dissecting ponzi schemes on ethereum: identification, analysis, and impact," *arXiv preprint arXiv:1703.03779*, 2017.

[22] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, and G. Rosu, "Kevm: A complete semantics of the ethereum virtual machine," Tech. Rep., 2017.

[23] CryptoNews, "CCN: Bitcoin, Ethereum, NEO, ICO & Cryptocurrency News," https://www.cryptocoinsnews.com/, (Accessed on 11/25/2017).

[24] Y. Zhang, S. Egelman, L. Cranor, and J. Hong, "Phinding phish: Evaluating anti-phishing tools." ISOC, 2006.

Authorized licensed use limited to: NED UNIV OF ENGINEERING AND TECHNOLOGY. Downloaded on July 07,2022 at 18:57:09 UTC from IEEE Xplore. Restrictions apply.