School of Informatics

Research Methods In Security, Privacy, and Trust
Detecting Ethereum Smart Contract Security Loopholes

2187344
January 2022

Abstract

Date: Thursday 20th January, 2022

Supervisor: Lorenzo Martinico

1 Introduction

During the predawn of the 21st century, we glimpsed a rapid technological and economic development that brought us face to face with a new technology called the blockchain. In recent years, blockchain and foremost cryptocurrencies gained much attraction due to the high monetary gains, which seem astronomical compared to traditional stock markets. It is a matter of the fact that this growth has acted as a catalyst for the technological and research eruption we have been experiencing lately. By the end of 2021, the total cryptocurrency market capitalization was 2 Trillion US Dollars[], featuring more than 16000 projects[]. Initially, the blockchain was proposed to facilitate the transfer of value completely decentralized and trust-framed over a network of peers. This was Satoshi's Nakamoto Bitcoin[] that came into existence in 2008. Since then, the research and technological trends have shifted into second-generation blockchains. Among them, the most dominant is Ethereum[]. Ethereum is a general-purpose blockchain, providing an open platform for individuals to build their applications on top of it. Their applications, i.e., Smart Contracts, are pieces of code that run decentralized on the Ethereum network. This allows developers to get involved in this new technology, creating an entirely new industry. Ethereum's ecosystem flared into a broad spectrum of Smart Contract applications, including, but not limited to, financial apps, games, digital art and music, digital voting, and patent registration.

Smart Contracts have balance measured in Ether and persistent private storage. The Smart Contract's code can manipulate changes in the program's variables and storage. On Ethereum, Smart Contracts code is in EVM[] (Ethereum Virtual Machine), a Turing complete stack-based bytecode language spanning 144 OP codes[]. Nonetheless, developers can define their code in a high-level language such as Solidity[], compiled to EVM bytecode afterward. A new transaction invocation can cause the execution of the contract's code by receiving inputs and producing outputs. Typically, when a user wants to trigger a Smart Contract needs to create a new transaction and pay some fees[]. Transaction fees depend on the code being executed on the Ethereum network.

Beyond any doubt, the Ethereum blockchain is used to manage digital assets reflecting a considerable value. Hence Smart Contracts gained the interest of malign entities. Numerous attackers perform attacks on the Ethereum applications, with the ultimate goal of stealing Ether from them. One of the most severe attacks was the infamous DAO[]. In more detail, in 2016, an autonomous decentralized organization was founded to direct a venture capital fund. This organization was formed on the Ethereum blockchain, and over 11000 investors deposited into a Smart Contract over 150 Million US Dollars[] considering the Ether price back then. Then, an unprecedented attack[] was performed on that Smart Contract, resulting in 50 Million US Dollars loss. Another critical incident happened with the Parity Wallet[], where the attack [] led to freezing 150 Million US Dollars, in terms of Ether, impermanently.

Nevertheless, it is typical for any piece of code to have bugs. Likewise, we expect Smart Contracts to behave in the same way. Potentially, five main reasons make Smart Contracts vulnerable. We might blame developers for not wholly understanding the blockchain development stack[smart check]. Also, Solidity is a relatively new language with many limitations and challenges. Thus, it is hard for developers to use it[smart check]. We consider blockchain immutability as another non-helping characteristic. Any application deployed on the Ethereum network can not be modified, so software fixes are not immediately doable[contract fuzzer]. A public blockchain allows financially motivated attackers to use their online pseudonymity to exploit any software bug[smart check]. Finally, we cannot control the Smart Contract execution environment since the network runs in a decentralized fashion[smart check]. For this reason, Smart Contract developers need to detect possible vulnerabilities in their code before going live. A Smart Contract Auditing tool would benefit both users and developers. A developer might use the tool to detect any code issues before deploying the Smart Contract. At the same time, users can utilize the tool to check if the Smart Contract they are depositing Ether is safe and does behave maliciously.

Indeed, such tools came into existence in 2016, with the first one being OYENTE[]. Since then, academic researchers have invested time and effort in expanding this research field, developing a plethora o tools. Some of them are general, trying to detect any vulnerability, while some others focus on a few of them. Also, they employ different analysis techniques, such as Static Analysis, Dynamic Analysis, Symbolic Execution, Fuzzing, and Machine Learning. Furthermore, researchers attempted to document and classify many of these vulnerabilities according to their behavior or functionality.

The contribution of this work is three-fold. Initially, we document and classify Smart Contract vulnerabilities reported across the most prominent studies. Next, we present the research trends of developing a Smart Contract vulnerability framework throughout the years, giving an overview of the techniques used and directions taken. Finally, we elaborate on the result reliability and scalability of the most paramount frameworks.

The structure of the following literature review has as follows ...

2 Methodology

In this literature review, we explore the most prominent research attempts towards creating an effective vulnerability detection tool for Ethereum Smart Contracts. The method used to filter relevant studies comprised both bottom-up and top-down strategies. Their combination significantly accelerated identifying adequate quality papers to include in this literature review.

During the bottom-up stage, we sought several papers related to frameworks that identify Smart Contract vulnerabilities. To achieve that, we employed trustworthy academic search engines such as *Google Scholar* and *IEEE Explorer*. We noticed that most of the retrieved writings had a joint primary related work, which guided us to discover this research topic's essence paper[1].

Subsequently, we aimed to find any derivative studies related to the paper arisen in the previous phase. To do so, we used a graph representation tool[2] that links relevant papers. This mechanism allowed us to identify remarkable research articles rapidly. Afterward, we manually inspected the search results and included the most reliable in our under investigation list.

Namely, we collected forty-seven papers, but we only consider nineteen of them in this literature review. The selection criteria span the context of the studies and their research contribution. We measure their contribution according to their citations and release year. According to their publication year, studies are assigned a weight ranging from 1 to 4. Papers in the span of 2016-2017, 2018, 2019, 2020 receive 1, 2, 4, 8 points, respectively, for each citation they hold. Using this metric, we evaluate the studies shown in Table 2.

Study	Year	Cit.	Score	Ref.
Making Smart Contracts Smarter	2016	1445	1445	[1]
A Survey of Attacks on Ethereum Smart Contracts (SoK)	2017	1176	1176	[3]
Securify: Practical Security Analysis of Smart Contracts	2018	436	872	[4]
VerX: Safety Verification of Smart Contracts	2020	103	824	[5]
ZEUS: Analyzing Safety of Smart Contracts	2018	399	798	[6]
Finding The Greedy, Prodigal, and Suicidal Contracts at Scale	2018	357	714	[7]
SmartCheck: Static Analysis of Ethereum Smart Contracts	2018	291	582	[8]
Formal Verification of Smart Contracts	2016	525	525	[9]
ContractFuzzer:Fuzzing Smart Contracts for Vulnerability Detection	2018	233	466	[10]
Slither: A Static Analysis Framework For Smart	2019	109	436	[11]
MadMax:Surviving Out-of-Gas Conditions in Ethereum Smart Contracts	2018	213	426	[12]
Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts	2019	91	364	[13]
teether:Gnawing at Ethereum to Automatically Exploit Smart Contracts	2018	173	345	[14]
Vandal:A Scalable Security Analysis Framework for Smart Contracts	2018	154	308	[15]
ReGuard: Finding Reentrancy Bugs in Smart Contracts	2018	122	244	[7]
Ethereum Smart Contracts: Vulnerabilities and their Classifications	2020	12	96	[16]
ETHPLOIT:From Fuzzing to Efficient Exploit Generation against Smart Contracts	2020	10	80	[17]
GasFuzzer: Fuzzing Ethereum Smart Contract Binaries to Expose Gas-Oriented Exception Security Vulnerabilities	2020	8	64	[18]
SoliAudit: Smart Contract Vulnerability Assessment Based on Machine Learning and Fuzz Testing	2019	13	52	[19]

Table 1: List of papers that are analyzed in this literature review, sorted by their score

3 Literature Review

3.1 Smart Contract Attack Surface

One of the most important aspects of Smart Contract vulnerability detection tools is understanding and realizing models that precisely describe a vulnerability. Different authors have unique approaches on how to classify potential faulty code. For example, according to their overall behavior, Nikolic et al. [Finding greedy ...] categorize Smart Contract into Greedy, Prodigal, and Suicidal. Namely, each category represents an abstract representation of a group of more explicit vulnerabilities. On the other hand, some other authors prefer to distinguish vulnerabilities based on the domain they affect. For example, a few studies, like GasFuzzer [gas fuzzer], choose to examine only gas-oriented vulnerabilities. Subsequently, it is abundantly clear that there is a broad spectrum of Smart Contract vulnerabilities, and there is not a single universally accepted classification of them. In 2017 Atzei et al. [sok] published a survey including the most well-known security pitfalls of the Ethereum Smart Contracts. In more detail, they report twelve points, taxonomizing them into three categories (Solidity-Level, EVM-Level, and Blockchain-Level). Four years and over ten million blocks later, new vulnerabilities arose. Hence, within the context of this literature review, we have collected reported vulnerabilities across different studies and present a thorough list of them, trying to categorize them into seven groups.

We categorize vulnerabilities according to their domain. In the first place, we noticed that a

Class	No.	Vulnerability Name
Syntax Violations	1	Interface violation
	2	Compiler version inconsistency
	3	Variable visibility modifiers
	4	Function visibility level
	5	Argument validation
	6	Costly coding scheme
	7	Redundant Fallback Function
	8	Code format/style violation
Internal Vulnerabilities	9	Ether Freezing
	10	Destroybale Contract
	11	Internal variable access control
	12	Contract balance access control
	13	Exposed secret
External Vulnerabilities	14	Reentrancy
	15	Denial of Service
	16	Poor exception handling
	17	Gasless Send
	18	Non isolated external calls
	19	Delegate calls
Arithmetic Vulnerabilities	20	Integer under/over flow
Gas related Vulnerabilities	21	Denial of Service
	22	Integer under/over flow
	23	Unbounded mass operations
Transaction related Vulnerabilities	24	Transaction ordering dependency
	25	Transaction state dependency
Block related Vulnerabilities	26	Block state dependency

Table 2: List of vulnerabilities according to their domain

few frameworks tend to include in their auditing report warnings related to Smart Contract syntax or related good programming practices. Thus, we included a category to wrap up all of these suggestions. Furthermore, we have classes referring to internal, external, and arithmetic bugs. Finally, we introduce two more categories to enclose transaction and block dependency exposures. Ultimately, we report twenty-two vulnerabilities over these domains (Note: a vulnerability might be part of multiple domains). Table 2 depicts the category and vulnerability mappings. The following subsections describe each vulnerability in more detail.

Interface Violations: Interface violations refer to implementing particular digital assets on the Ethereum blockchain without fully aligning with the defacto community accepted requirements. For example, ERC-20 tokens [eth website to erc20] need to expose a particular API interface to help third-party services communicate with their Smart Contract flawlessly. If a developer decides to implement a non-ERC-20 compliant token, it might be considered suspicious and thus malicious.

Compiler version inconsistency: Solidity source code files include a header line stating the versions of compiler they can be complied with. A similar to Javascript notation can be used to indicate the minor compiler version (caret character). This notation can produce execution inconsistency in different execution environments. For this reason, it is better to declare a stable compiler version precisely. Nevertheless, we can argue with this proposal from Tikhomirov et al. since using a stable compiler version might make a Smart Contract obsolete and incapable of running on newer EVM versions.

Variable visibility modifiers: Many developers migrating from traditional programming languages mistake the *private* variable modifier as not visible to others. This is a misunderstanding since every Smart Contract state variable is visible to anyone. The term *private* means that other Smart Contract instances can not access this variable.

Function visibility level: In Solidity, a function can have three visibility levels. These are external, internal, and public. Likewise to the previous coding pitfall, many developers might misinterpret the meaning of these modifiers and use them wrong.

Argument validation: It is a common practice to adopt for every Smart Contract function. For example, when we receive a string of a maximum of eight characters long, we should check that its length is valid. However, this practice can require more computational complexity and yield more gas fees. It should be used when needed.

Costly coding schemes: In Solidity, different data structures have different gas costs, but they can achieve the same result most of the time. For instance, a developer should use pure bytes instead of using a byte array because they have lower gas fees. Moreover, loops a the most common coding scheme to avoid since a loop of an unknown length comes with high execution cost.

Redundant Fallback Function: Solidity supports a fallback/receive function triggered when no function signature matches the transaction data. This function deposits the incoming value to the contract balance when a transaction has value but no data. Similarly, it handles any transaction that has unknown function signatures. A fallback/receive function should be implemented only when needed; otherwise, it is redundant.

Code format/style violation: Smart Contract developers should use tidy and easy-to-read styling patterns throughout the code.

Ether Freezing: A Smart Contract, as mentioned earlier, has its balance of Ether. Different types of applications require the user to deposit Ether to participate in a lucky draw or an online game etc. Some Smart Contract will accept deposits, but under specific conditions, it might lock Ether forever, acting as a black hole. Something similar happened in 2017 with Parity wallet Smart Contract. The bug occurred because the Parity wallet transferred funds using an external library. That library was accidentally removed from the blockchain, and thus all Ether in Parity wallet became inaccessible. Further, according to Nikolic et al. another case where funds can be locked in a Smart Contract is when the contract is removed from the blockchain, and some users continue to send Ether to it. These Ether will be lost as well.

Destroybale Contract: Smart Contracts, out of the box, give the ability to their owners or an authorized entity to destroy them. The funds enclosed in the Smart Contract will be returned to the owner by default. However, if an arbitrary account can destroy the Smart Contract, it alarms a vulnerability. This abnormal code behavior might have been maliciously constructed and helped an attacker harvest any Ether in the Smart Contract's balance when combined with other attacks.

Internal variable access control With the term internal variable access control, we refer to insecure Smart Contract implementation that allows an adversarial entity to alter the internal state of the Smart Contract. This kind of behavior can be combined with the earlier mentioned Destruction vulnerability. For instance, an attacker leverages his ability to alter the Smart Contract owner variable before the Destruction. Besides that, attackers might straight forward access the contract's private variables value or logic, revealing an important secret.

Contract balance access control Like internal variable access control, a Smart Contract that can be invoked to transfer money to an arbitrary account appears questionable. We define arbitrary accounts as addresses that have never deposited Ether into a Smart Contract but can withdraw value from it.

Delegate calls In Solidity, developers can call third-party Smart Contracts already deployed on the Ethereum blockchain. This technique is called delegate call and has the advantage of reducing the initial gas costs required to get an application online. However, if the third-party library is exposed or destructed, it directly affects the execution of our application. Thus, a delegate call might lead to security pitfalls.

Reentrancy We consider the function of a Smart Contract reentrant when it executes an external call to a contract that maliciously calls back again the caller function. In other words, there is a recursive entrance to the initial function. This phenomenon can be problematic, especially in cases where the code being executed leads to Ether withdrawals. This issue became famous during the DAO[] hack back in 2016.

Denial of Service Denial of Service is a well-known attack in many systems. Essentially, the attacker removes the Smart Contract availability to other users. This is attainable when a condition within the Smart Contract's code relies on the result of an external call or input. The callee might permanently fail to respond to that call, preventing the caller from ending its execution.

Poor exception handling Exception disorder is a common problem in Smart Contracts and depends on poor exception handling when a contract calls another. The failure to call another contract function may result in different errors, and various handling is needed. Instead, most developers do not check for these exceptions, and subsequently, these transactions fail. This can be problematic in several cases, primarily when Ether is in transit, and any code executed until the fail point will revert.

Gasless Send According to Consesys[], the send() function is wrong since it only forwards 2300 gas. This can cause problems if the fallback function of another Smart Contract is costly to execute and fails. Further, gas prices might change in the future, and 2300 will not be enough to cover any functionality that can be performed today with 2300 gas units.

Non isolated external calls Having several call invocations in a loop is not a good idea. This is because the failure of one call might lead to canceling all preceding calls in the loop. Hence we need each call to be isolated. For instance, good practice in withdrawals is the pull mechanism[], where a user initiates the transaction for withdrawing Ether to his address. Hence, its transaction will not affect any other transactions.

Integer under/over flow In Solidity, integer over/under-flows can happen. For example, a uint8 integer can take values from 0 to 256. If the current state of a uint8 variable is 0 and we subtract 1 from it, its value will rotate to 256. Nevertheless, this issue is being resolved in newer Solidity versions [].

Unbounded mass operations Unbounded mass operations mainly refer to loops. Iterating over a large data structure, like arrays, might result into out of gas exceptions. Each block allows only a specific amount of gas, and when exceeded, we get an exception, and our transaction fails.

Transaction ordering dependency Transaction ordering dependency is another vulnerability to think about. Some transactions are essential to execute in a particular order to produce the desired outcome. For example, a user wants to sell tokens to a smart contract for Ether using a contract-defined rate. The maligned Smart Contract owner issues an order to reduce that rate and ensures his order is executed first. This results in tricking the user into accepting a different exchange rate.

Block state dependency Block state includes information such as block number and timestamp. This information is insecure to use as input for the Smart Contract functions. Notably, it is not secure to use them as a source of randomness since they are predictable, and any adversarial entity might use them for its interest to simulate the outcome of a tranasction.

3.2 Static Analysis

Static analysis is an automated method of analyzing software without executing its code. This is achieved by only considering the behavior of statements and declarations throughout the code. Static analysis frameworks achieve that by employing formal methods such as data flow analysis, model checking, inference rules, semantic language analysis, and symbolic execution.

Using the data flow analysis, tools can produce a control flow graph. This graph depicts how a program can be separated into basic blocks[] and how each block connects, forming different execution paths. In addition, using model checking, we can observe if a program meets certain conditions and has a finite state. Inference rules are first-order logic statements representing any code dependencies and logical actions. Besides that, we use semantic analysis to reflect how a programming language works. In our case, we are interested in Solidity bytecode semantic rules formalization. Using these semantics, we can verify a Smart Contract's correctness. Ultimately, a more practical approach comes with symbolic execution. Symbolic execution explores possible execution paths assuming symbolic inputs. The final result of the execution generates a set of expressions and constraints involving the program variables. We feed these constraints into a constraint satisfiability solver to generate concrete inputs for the paths found.

One of the first works on this topic was proposed back in 2016 by Luu et al.[], presenting the first of its kind tool called OYENTE. OYENTE operates using the Smart Contract's bytecode, and it is mainly based on symbolic execution. The researchers preferred to use symbolic execution since it is easier to reason about a program path-by-path. The engine of this framework extracts the CFG out of the Smart Contract bytecode, distinguishes valid execution paths, and eventually uses the Z3 Bytevector solver[] to generate tangible values validating any vulnerabilities. We note that OYENTE evaluated using 19366 Smart Contracts with 350 seconds per contract average analysis time. Also, this tool yields many false positives. Follow-up work has been proposed with the tool named Manticore[]. Mossberg et al. take a different approach to path exploration, defining three states that a Smart Contract can be. They suggest that a contract can be Ready, Busy, or Terminated. In particular, their framework tries to explore the different states that a Smart contract can be during its execution using various heuristics. Afterward, they feed to each Ready state a symbolic transaction. Manticore is a modular framework that allows different SMT solvers to be used. Regarding this framework evaluation, the authors tested it using only 100 Smart Contracts, which is not a representative amount of contracts in arguing about the tool's scalability. Another drawback of this tool is that it only achieves 66% code coverage on average, which is relatively low. MAIAN[] is another symbolic execution framework proposed by Nikolic et al., having good true positive results in detecting the Contract balance access control and Destroyable contract vulnerabilities. Furthermore, Krupp et al. suggested their tool, teEther, based on the same logic. Nevertheless, their work is inadequate since they analyzed 815 contracts out of the 33195 they collected. They stated that only 815 contracts exposed critical paths. According to their assumptions, a critical path leads to Smart Contract external influence. Hence, their result granularity is very strict.

On the other hand, Securify[] utilizes data flow analysis, model checking, inference rules, and semantic language analysis to detect vulnerabilities. In more detail, this framework takes as input EVM bytecode and a set of predefined security patterns. Next uses the semantic facts of the EVM, including some CFG data, to match any security issues found according to the given patterns. One of the drawbacks of this tool is that it assumes the bytecode instruction (sstore), malicious since it can not realize it correctly using semantic analysis. This fact might lead to false positives. Eventually, Tsankov et al. evaluated Securify using 24594 Smart Contracts. Comparing this kind of tool with symbolic execution-only tools, we observe that it is more successful with fewer false positives.

An alternative school of thought within the static analysis sphere leverages abstract interpretation to analyze Smart Contracts. Well-known studies following this method are Zeus[], SmartCheck, Slither, Vandal, and MadMax. Initially, they take as input Solidity source code or bytecode and translate it into an intermediate representation. Such representations can be in XML form [smart check] or LLVM[zeus] code. Next, similar to Securify, these tools take a set of predefined policies as input and translate them into an intermediate representation. The final step is familiar with the previously described tool where an SMT solver tries to analyze points in the code where the policy verification predicates need to be asserted. Zeus, Slither, and Smart check are evaluated with 22493, 4600, and 1000 Smart Contracts, respectively. The three of them have significantly better results than Securify, with Zeus reporting only a 2% false-positive rate. MadMax, which is based on Vandal framework, is a vulnerability detection tool specifically designed to catch out-of-gas exceptions. Surprisingly, MadMax can analyze the whore blockchain, snappy, with an average analysis time of 8 seconds per contract and a reasonably low false-positive rate.

3.3 Dynamic Analysis

In contrast to Static analysis, Dynamic analysis aims to detect any code vulnerabilities during the runtime of a program. It is a matter of the fact that Dynamic analysis produces almost zero false positives since every explored path is being executed. Therefore, we know for sure that it is reachable. There are two main approaches when using Dynamic analysis, instrumentation, and fuzzing. In the Smart Contract auditing research domain, most works are fuzzing oriented.

The concept of fuzzing is inspired by random input generation for programs expecting to crash them and reveal potential vulnerabilities. A fuzzing engine generally comprises the input generator and the monitor that detect errors during the program run time. In addition to pure random value generation, we have mutation-based fuzzing, aiming to expose a more narrow domain of bugs. Fuzzing is a contemporary practice taken when analyzing Smart Contracts.

ContractFuzzer[] is a state-of-the-art framework transcending the complex and unsolvable mathematic constraints of Static Analysis. GasFuzzer takes a Smart Contract's application binary interface (ABI), extracts the function signatures, and uses them to generate possible inputs. Afterward, the code is executed using the generated inputs, while executions logs are dumped into an external log pool. Eventually, predefined error detection oracles pass through the recorded logs to identify potential bugs. It is worth noting that during the execution of the Smart Contracts, researchers initiated transactions coming from different types of accounts (owner address, external address, attacker Smart Contract address). This is something we could not accomplish using Static Analysis.

Building on top of Contract Fuzzer, Ashraf et al. created Gas Fuzzer[], a gas-oriented fuzzing tool. This tool employs two different methods of fuzzing, gas-greedy, and gas-leveling. The gas-greedy strategy generates random transactions putting them in a pool. The fuzzing engine takes one, mutates it, and then executes it. If the mutated transaction gas is more than the pre-mutated transaction, the first one is placed back to the transaction pool. In the meantime, the Gas Fuzzer checks the execution fingerprint of the mutated transaction for potential vulnerabilities. This process runs iteratively until a predefined runtime timeout reaches. On the other hand, gas leveling introduces the concept of delivering to the fuzz engine transactions consuming gas of different levels. This strategy seeks to report the code coverage reached using different amounts of gas. Once coverage converges to its maximum, the tool checks for bugs using the execution log.

Some other fuzzing extends Static Analysis techniques to achieve their goals. Namely, Reguard, a reentrancy detection framework, uses Static Analysis to translate into intermediate representation before generating any random values. In more detail in this work, researchers attempt to translate Solidity into C++ to use an out-of-the-box fuzzing engine, AFL[]. Eventually, they check for bugs using the execution log, which they feed into automata describing the reentrancy vulnerability.

3.4 Future Directions

4 Summary & Conclusion

References

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

[2] Find and explore academic papers. `https://www.connectedpapers.com/`. Accessed: 2021-11-04.

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International conference on principles of security and trust*, pages 164–186. Springer, 2017.

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

[5] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1661–1677. IEEE, 2020.

[6] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *Ndss*, pages 1–12, 2018.

[7] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68. IEEE, 2018.

[8] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 9–16, 2018.

[9] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*, pages 91–96, 2016.

[10] Bo Jiang, Ye Liu, and WK Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 259–269. IEEE, 2018.

[11] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE, 2019.

[12] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018.

[13] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.

[14] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1317–1333, 2018.

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

[16] Zulfiqar Ali Khan and Akbar Siami Namin. Ethereum smart contracts: Vulnerabilities and their classifications. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 1–10. IEEE, 2020.

[17] Qingzhao Zhang, Yizhuo Wang, Juanru Li, and Siqi Ma. Ethploit: From fuzzing to efficient exploit generation against smart contracts. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 116–126. IEEE, 2020.

[18] Imran Ashraf, Xiaoxue Ma, Bo Jiang, and Wing Kwong Chan. Gasfuzzer: Fuzzing ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities. *IEEE Access*, 8:99552–99564, 2020.

[19] Jian-Wei Liao, Tsung-Ta Tsai, Chia-Kang He, and Chin-Wei Tien. Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 458–465. IEEE, 2019.

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