A Meta-Analysis of Smart Contract Vulnerabilities and Analysis Tools

Adrian Lopez, Nathan Luksik, Pranay Jain

24 April 2022

1 Introduction

Smart Contracts have facilitated the invention and proliferation of modern advancements in financial technology, including Decentralized Finance and Decentralized Autonomous Organizations. Ethereum introduced the implementation of smart contracts in public permissionless blockchains. A smart contract on Ethereum is a stateful program with a designated address on the blockchain. It includes executable code and mutable data that reflects the contract's current state. The bytecode of the program and its state are replicated by all the miners and validators in the network. A useful analogy for Ethereum, and other smart contract-enabled blockchains, is that of a *distributed state machine*. The entirety of the blockchain can be thought of as a distributed system with a shared state that can change from block-to-block.

Transactions are cryptographically-signed instructions from accounts - where accounts could belong to either an individual's wallet or a smart contract. Once a transaction is included in a block, the miners/validators execute the relevant code to update the state. The specific rules for updating state are concretely defined by the blockchain; for example, in Ethereum the specific rules of changing state from block to block are defined by the Ethereum Virtual Machine (EVM) (4). As we will discuss further, updating the state for a transaction may involve executing the code for one or more contracts, which can lead to behaviour and attack vectors that are hard to predict.

A major challenge with deploying smart contracts is their immutability. Contract creation is essentially a special type of transaction on the blockchain. Since transactions are fundamentally immutable and irreversible in a blockchain, it is not possible to edit or remove a contract that contains bugs. This is particularly problematic where certain contracts accumulate large amounts of the cryptocurrency in their accounts over time and are eventually exploited leading to huge losses. The website rekt.newscompiles a list of smart contract exploits and their corresponding monetary losses - the most largest recent hack is the Ronin bridge hack that caused a loss of $624 Million (based on Ethereum's current value of approx. $3,000).

It is therefore imperative for developers to be able to detect and fix bugs in their contract code *before* deployment. This is an active area of research. There have been efforts to define various classes of known bugs, which has the benefit of making analysis more structured, and also detect other related bugs. There also exist several analysis tools that automate the process of auditing contract code and detecting various bugs. In this paper, we review the state-of-the-art methodologies and tools for detecting and resolving smart contract bugs. We compile our research into actionable steps such that a developer can determine the best options to analyze their code before they deploy it on the blockchain.

The rest of the paper is structured as follows. Section 2 discusses the broad categories of vulnerabilities in smart contracts and gives a deeper dive into four kinds of vulnerabilities that can be potentially most dangerous. Section 3 discusses the state-of-the-art in analysis tools for contracts, along with the advantages and limitations of each tool. As it turns out, no current tool reliably covers all kinds of bugs, so a combination of tools may be required for best coverage. Most of the analysis in our paper concerns with vulnerabilities and analysis tools for the *Solidity* (5) language, which is the dominant language for writing smart contracts for Ethereum. In section 4, we will summarize our findings and experiments with the different tools and provide the best options for developers to use.

2 Smart Contract Vulnerabilities

There are many known bugs and vulnerabilities in smart contracts. While some are language-specific (eg. integer overflow bugs), others are relevant to blockchain design. Zhang et al (15) propose a methodology for categorizing smart contract bugs into 9 main categories. In the paper, they divide these furhter into 49 further subcategories; these categories are defined based on comprehensive analysis of a dataset of bugs. In this section, we focus on some of the more dangerous and hard to detect bugs.

2.1 Mishandled Exceptions

There are several ways for one contract to call another contract (such as via `send`) or a function of another contract (e.g. `anotherContract.aFunction()`). If there is an issue with the call, such as the caller contract not sending enough gas or exceeding the call stack limit, then an exception is raised in the callee contract. The callee contract then terminates, reverses its state, and returns `false`. However, the exception in the callee contract may or may not get propagated to the caller contract, depending on how the call was made. For instance, if a call is made via `send` instruction, then the caller contract needs to explicitly check the return value to verify whether or not the call has been correctly executed. This inconsistency regarding exception propagation policy leads to man instances where exceptions are mishandled.

Consider, for example, a contract where a particular wallet address is assigned as the "leader", and to take the leader's place, other addresses will pay some amount more than

```
function oddOrEven(bool yourGuess) external payable returns (bool) {
    if (yourGuess == now % 2 > 0){
        uint fee = msg.value / 10;
        msg.sender.transfer(msg.value * 2 - fee);
    }
```

Figure 1: Example of a Timestamp Dependent Contract

the previous leader paid to attain the position. The leader, upon getting dethroned, profits from the difference between the price he paid to the previous leader and the price the next leader pays to him. If this contract does not check the result of the transaction, then if there is some exception and the transaction does not go through, the outgoing leader will be replaced without any compensation.

A malicious user can take advantage of this by deliberately exceeding the call stack limit, causing the `send` instruction to fail. The EVM limits the call-stack depth to 1024 frames, and this depth increases by one when a contract calls another via the `send` instruction (7). Thus, the malicious user can have a contract call itself 1023 times before sending a transaction to this vulnerable contract to claim leadership over the current leader, resulting in that leader not receiving any payment.

2.2 Timestamp Dependency

Timestamp dependency is introduced in a smart contract when the outcome of some function execution depends on the value of the timestamp of the transaction. There are many use cases when one may need this functionality: for example, in an auction, we accept a bid only if the bid is submitted before the auction end time.

When introducing a new a block, the miner sets the timestamp for the block. This block and its timestamp are of course checked by other miners for approval, so the timestamp is normally set as the current time of the miner's local system. However, a malicious miner can actually vary this by up to roughly 900 seconds without getting other network validators to reject its blocks (10). So, a miner may choose different block timestamps to manipulate the outcome of timestamp-dependent contracts. Smart contract that use the block timestamp as a triggering condition to perform some critical operation, e.g. transferring Ether, or as the source of entropy to generate random numbers, are considered the most at risk for timestamp dependency attacks.

For example, consider the following simple function in Figure 1. In the example, the `now` in the if-statement refers to the timestamp that can be manipulated by a miner, who might predetermine the block timestamp to be odd or even, thereby deciding whether or not the if-statement returns true, and thus whether or not the transfer ever occurs. The only solution to this issue is to not rely on a block timestamp to perform critical operations in smart contracts.

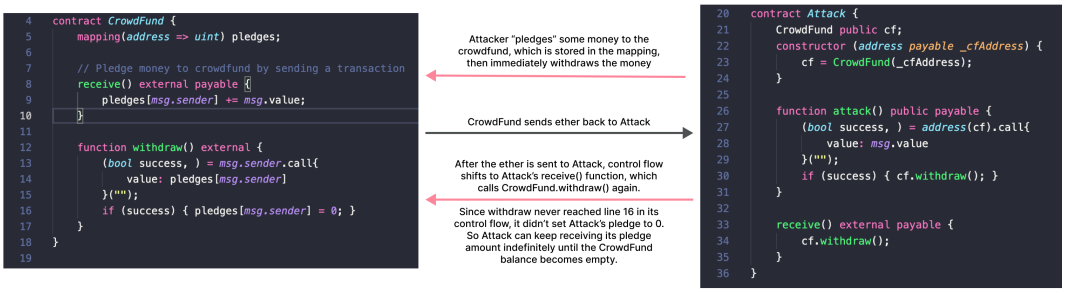

Figure 2: An example of Reentrancy Attack on a simple crowdfund contract. Example inspired by (15) and adapted to the latest version of Solidity.

2.3 Reentrancy Vulnerability

A smart contracts is just a piece of executable code with a corresponding public address. A contract typically has a default `receive` function, which is executed any time it receives ether in a transaction; it may also expose other public functions using the `external` modifier, that can be executed by other smart contracts as special transactions. Informally, a reentrancy bug is a blockchain parable of "infinite loop" bugs in normal software programs – essentially, a malicious contract may call a public function in such a way that it can "re-enter" the same function and execute it repeatedly.

Consider the example in Figure 2 on a simple contract for a crowdfunding campaign. Any individual can pledge money by sending ether to the `CrowdFund` contract, which calls the `receive` function. The figure shows how an attacker may publish another contract called `Attack` that can effectively drain the entire balance of the crowd fund by mounting a reentrancy attack.

This vulnerability was behind the famous `TheDAO` attack (3) which allowed an attacker to drain 3.6 million ether (worth over $10 billion based on current approximate value of $3,000 for 1 ether) from their contract. Due to the notoriety of this attack, fortunately this attack is now widely studied. As a result, modern analysis tools are able to detect many occurrences of this bug. However, they're still not foolproof; there are several recommendations to developers to mitigate such attacks. The first is to always update the internal state of the contract before making any external calls. Another fix is to use the `send` function (instead of `call`), that does not transfer control flow to the destination when executed. Finally, there also exist libraries that provide **Reentrancy Guards**, which are simple plug-and-play function modifiers that protect from reentrancy attacks. An example of this is the OpenZeppelin `ReentrancyGuard`.

2.4 Transaction Order Dependency

When a transaction is requested on a blockchain, it is sent as an RPC request to all the miners in the network. It is up to the miners to decide what transactions are added to a block

and what ordering the transactions appear in *within* the block. Typically, miners choose an ordering that maximizes their block rewards (which depends on the transaction fees). However, this is not a requirement as transaction ordering is not enshrined in the consensus rules. This means transactions that happen after [1] a transaction may end up being executed earlier. This can lead to serializability violations.

This enables two possible vectors for attacks. Firstly, a malicious miner may purposely publish ambiguous transactions along with the legitimate ones to cause long-term inconsistencies (14). Another attack (which is more prevalent) is called *frontrunning*. It involves malicious parties to observing the network for certain transactions of interest and then immediately publishing their own (competing) transactions that may be confirmed earlier, leading to favourable outcomes for the attacker. This is a significant challenge to fairness and equity of outcomes, as it favours participants with greater resources, and is subject to discussions in the Ethereum developer community (11).

Contracts have a Transaction Order Dependency (TOD) vulnerability if such attacks can cause potentially dangerous outcomes. A simple example is illustrated in Figure 3. Consider the code for a simple contract that facilities P2P trading of some digital goods (similar to eBay). Say a user looking to purchase some digital asset calls the `buy()` function to purchase at the price currently specified by the `price` storage variable. Then, either maliciously or ignorantly, the contract owner calls `setPrice()` and increases the value of the `price` variable, and then sends this transaction with a higher gas fee than the buyer. The owner's transaction is likely to be confirmed first, so when the buyer's transaction is subsequently mined, the `buy()` function is using the updated (increased) price, not the original price the buyer bid on.

2.5 Event Order Vulnerabilities

As discussed in 2.3, smart contracts typically expose one or more public functions that can be invoked by other contracts. These function may have many "calls" (which are essentially transactions) invoked simultaneously, and the order in which the execution occurs depends completely on miners. Consequently, one may view the state of a contract as a concurrent data structure. Kolluri et al (9) define Event Ordering (EO) bugs as errors resulting from violation of the desirable concurrency properties of contract state. Formally, an event-ordering (EO) bug exists in a contract if two different event traces, consisting of the same set of concrete events, produce different outputs.

There are two main properties that are considered for EO bugs. The first property is **Linearizability** that may arise when contracts query "off-chain" oracles that respond asynchronously via callbacks. While off-chain oracles afford many great features, they can

[1] Technically, there is no "happens before" relationship since the original transaction was never added to a block. However, in a public blockchain, transaction *requests* are visible to all participants even before they are committed. This suggests the need for a more apt parable for the "happens before" relationship in the context of blockchain transactions, though that is outside the scope of this report.

```
function buy() returns (uint256) {
    Purchase(msg.sender, price);
    return price;
}

function setPrice(uint256 _price) ownerOnly() {
    price = _price;
    PriceChange(owner, price);
}
```

Figure 3: Example of a Transaction Order Dependency in contract code

introduce linearizability violations if not handled correctly as callbacks may change the contract state in inconsistent ways. The second property is **Synchronization** which requires that certain type of commands (for example, sending money vs internal state update) should happen in a desirable order. Linearizability and Synchronization are both key properties of concurrent state machines and (9) is the first paper to formally define these properties in the context of smart contracts.

3 Smart Contract Analysis Tools

Smart contracts are quickly becoming mainstream as more users and companies are starting to adopt them. One of the biggest barriers to the widespread usage of smart contracts is their safety and vulnerability. As previously mentioned, smart contracts cannot be modified after deployment, which means bugs can't be mitigated and end up being very costly. More than hundreds of millions of dollars worth of Ether has been stolen and/or lost from just the Ethereum blockchain alone due to bugs which are hard to detect until it's too late.

In this section, we analyze the tools currently available, their strengths and shortcomings, and then provide a recommendation of tools that every smart contract developer should use before deploying their smart contract provided they have access to reasonable resources. In specific, we researched these tool's ability to be able to detect the bugs with respect to the Jiuzhou dataset (15), which is a open-source dataset of smart contracts that contain various vulnerabilities. In addition, we also discuss their success with detecting event-ordering bugs.

3.1 Performance Metrics

We refer to the performance metrics used in Zhang, P. et al (15) to analyze the performance of the detection tools analyzed below. Note that none of the tools except for EthRacer (9) is

able to detect event-ordering bugs.

Coverage: Coverage is defined as the percent of bugs detected divided by the total number of bugs. The tools were tested against 49 types of bugs defined in dataset defined by (15).

Precision: Precision aims to answer the question "Out of all predicted bugs predicted by each tool, how many of them are actually bugs?" Note this calculation, along with recall, is only with respect to the type of bugs this tool can detect (its coverage). It is calculated with the following formula showed below:

$$Precision = \frac{true\ positive}{true\ positive + false\ positive}$$

Recall: Recall aims to answer the question "Out of all known bugs, how many of them were detected with this detector?" It is calculated with the following formula showed below:

$$Recall = \frac{true\ positive}{true\ positive + false\ negative}$$

3.2 Remix

Remix (6) is an open source IDE that supports Ethereum based smart contracts written in Solidity. This IDE provides code checking and detecting bugs on compilation. It also supports many plugins to improve the Solidity development experience. For the 49 bugs mentioned in Zhang, et. al. (15), Remix (used without plugins) had the highest recall at 90%, however it did only have 49% precision and 22%. In other words, it was able to detect 11 out of the 49 bugs. This suggests that other tools should be used alongside this one since it has a narrow selection of bugs that it can detect.

3.3 Mythril

Mythril (2) is a security analysis tool for EVM bytecode, which is able to detect vulnerabilities in a wide variety of blockchain contracts that are EVM compatible. It is a Python-based library and can be downloaded via `pip`. It is also available as a plugin to the Remix IDE and is recommended to be used alongside Remix due to its ease of use and supplementary metrics. While it has a lower recall at 67% than Remix, but it provides greater coverage than Remix at 25% ($\frac{12}{49}$ bugs detected), and most importantly, a much higher precision at 76%. This means that things flagged by Mythril as bugs are more likely to true positives. Given the supplementary coverage and ease of use, we recommend its use alongside Remix.

3.4 Slither

Slither (7) is also an available plugin to the Remix IDE which makes it really easy to use for developers looking to test their smart contract for bugs. We **strongly** recommended to be

used alongside Remix and Mythryl due to its strong detection metrics on the Jiuzhou dataset. It has a much higher coverage than Remix at 55% ($\frac{27}{49}$ bugs detected), while having solid precision and recall metrics at 71% and 68% respectively.

3.5 Oyente

While Mythril, and Slither are all excellent tools for detecting bugs, they fail to detect transaction order dependence (TOD) bugs. These bugs are very hard to detect since they depend on a variety of different race conditions, which could be hard to simulate. Oyente (10) is a detection tool that aims to detect TOD vulnerabilities, along with other known bugs including Timestamp Dependence, Reentrancy and more. Oyente is able to detect some instance of TOD bugs and the authors claim that it can detect the `theDAO` bug.

However, under more comprehensive testing by (15), it turns out it has relatively poor performance with many bug classes. Oyente is able to detect 2 event ordering bugs, which gives them 6% coverage. On the Jiuzhou data set (15), they had 0% precision and 0% recall, suggesting that this tool is not very reliable and should be used with caution. Furthermore, we note that Oyente has not been updated in a long time and only supports Solidity versions up to 0.4 (the latest version is 0.8). This is a general concern with EVM-dependent tools since they need to be updated with each update of Solidity and the EVM. In our experiments, we had to convert our code to older versions of Solidity, so as things stand, it's not a very useful tool.

3.6 EthRacer

EthRacer (9) is a dynamic symbolic execution tool for Solidity smart contracts that is designed to detect Event Ordering bugs. EthRacer simulates and enumerates the path space for different event calls in a contract and establishes *happens-before* relationships between events in order to detect EO violations.

In their analysis of 10,000 smart contracts on the Ethereum chain, EthRacer flagged 7.89 % of smart contracts for synchronization violations. Additionally, EthRacer was able to flagged 11 % of all smart contracts linearizability violations. To demonstrate the severity of these types of bugs, out of the smart contracts flagged from the 10,000 smart contracts tested by this tool were known to hold more than $100M dollars worth of Ether throughout their lifetime. EthRacer was also able to flag all bugs detected by Oyente (78 in total) through these empirical experiments, while flagging an additional 674 more bugs that Oyente missed. This indicates that it is successful at finding TOD and EO bugs with a good success rate that beats Oyente.

However there is a cost limitation to EthRacer. It requires a fully synced Ethereum blockchain (at least up to the first 5.4M blocks) in order to be able to work. Furthermore, its execution time is significantly higher than other tools. Its 95 percentile of execution time is 18.5 minutes, while up to 5% contracts take longer than 150 minutes. As a result, it is

less of an active develop-and-test tool and may only be useful for testing once most of the development is complete. As such, we recommend that this tool be used by smart contract developers with the resources able to download the entire Ethereum blockchain and plan to deploy large, high volume smart contracts.

3.7 Other Tools and Scope for New Tools

There are several other analysis tools for detecting smart contract vulnerability. We focused on the above tools because they form the best complementary set for covering most known bugs. Some other tools that also have very good, albeit not the best performance are: Securify (13) which achieves low coverage but good recall and SmartCheck (12) which has a higher (but not best) coverage and good recall. Neither cover TOD and EO bugs. The tool `SERVOIS` (1) is able to detect a specific type of linearizability violation in smart contracts but does not scale as well as EthRacer (9).

We also found that error reporting for Timestamp Dependency bugs is not very reliable. From our experiments, we found that every command that considers the block timestamp is flagged as a Timestamp Dependency (note that this is the only way in smart contracts to get the current time). However, this is often unavoidable, eg. consider a simple auction that ends at some fixed date - the contract must reject all bids if `now` is later than the end date. Unfortunately, there's no way to distinguish a benign usage of the block timestamp with a potentially dangerous timestamp dependency using the tools currently available. Although Slither and Mythril provide some color coding for severity, it flags all TD bugs as yellow/moderate. This is a potential avenue to improve existing tools.

As such, there is still scope for making a general purpose fast and reliable smart contract analysis tool. It seems that most existing tools are not tested in controlled scenarios; ie. they are used to detect bugs on existing deployed contracts, but due to the scale of the task (with over 10,000 contracts), it is not possible to manually verify whether the detected bugs are actually serious bugs. As a result, there's no ground truth evaluation of these models, which shows how these tools perform relatively poorly (low precision/recall) on these simulated datasets of smart contract bugs. We recommend the work of Ghaleb et al (8) that proposes an open source tool for injecting known bugs into contracts that can be used to evaluate smart contract auditing tools under ground truth.

4 Experiments and Conclusion

After our experiments working with the various analysis tools mentioned above, we provide our analysis of the comparisons between these tools. Finally we conclude with our recommendations on best practices to consider when developing a smart contract on Ethereum.

4.1 Comparisons Between Analysis Tools

We judge the analysis tools based on three main categories: Usage feasibility, Coverage, and Output Usefulness. The first characterizes ease of use and how time- and computational-complexity of running the tool. "Coverage" refers simply to whether or not each aforementioned vulnerability can be generally found by a given tool. Finally, the "Output Usefulness" characterizes how well the tool helps to specifically identify vulnerabilities and help to locate and resolve the those which it finds.

Usage Feasibility

Remix scores very high in this category, as it is a simple browser IDE that provides rapid and real-time identification of errors. Mythril and Slither also have highly usage feasibility; they ship as pip packages and are easy to install, and also provide Remix integrations. Oyente is relatively worse since you need to set up a docker image for it. EthRacer is the most complex to set up - it requires setting up a docker image and downloading the entire Ethereum blockchain to run the tool.

Coverage

Refer to Table 1 for the exact coverage of each tool for the vulnerabilities discussed in Section 2. We found that Remix covers a wide variety of bugs and provides useful code checking, though it misses all the serious vulnerabilities discussed in section 2. It does integrate well with Mythril and Slither, however, which complement it with additional coverage. None of these tools cover TOD and EO bugs. Oyente covers TOD, Reentrancy and Timestamp bugs (along with others), though its reliability is questionable. EthRacer is very specialized to EO bugs, though it also detects some TOD and some instances of Reentrancy bugs as they are special cases of EO bugs (9).

Output Usefulness

Remix scores well in this category as it provides syntax highlighting and error location for the bugs that it does detect. Slither and Mythril both score very high here, as they not only identify vulnerabilities and their exact location, but also color-code them based on their severity (green = low, yellow = moderate, and red = high severity). This is very useful as the same error (eg. Timestamp dependence) can be both highly dangerous and totally benign in different scenarios. Slither also provides a direct link to a wiki explaining the bug and ways to mitigate it for each detected vulnerability. Oyente scores relatively poorly in this regard as it does not provide colour coding for error severity. Combined with its relatively high false positive rate, it may result in wasted effort on debugging bugs that are not important. We were unable to run EthRacer in our experiments due to lack of computational resources.

Tool | Bug	ME	TD	RE	TOD	EO
Remix	No	No	No	No	No
Mythril	Yes	Yes	Yes	No	No
Slither	Yes	Yes	Yes	No	No
Oyente	Yes	Yes	Yes	Yes	No
EthRacer	No	No	Partial	Yes	Yes

Table 1: A reference table depicting which tools are able to broadly identify which vulnerability classifications, including Mishandled Exceptions (ME), Timestamp Dependency (TD), Reentrancy vulnerability (RE), Transaction Order Dependency (TOD), and Event Ordering vulnerability (EO).

4.2 Recommendations

We strongly recommend that smart contract developers follow the procedure below before deploying their smart contracts:

- Use the Remix IDE for small teams or initial development stages of the contracts. Integrate Remix plugins for Mythril and Slither plugins for comprehensive bug coverage for all except the EO bugs. This is easy to do and has relatively low memory and computational resources requirements.
- For larger teams and bigger projects, Remix may not be viable as teams may require version control. We found that the Solidity plugin on Visual Studio Code also provides good linting and code completion features. Developers may want to implement Continuous Integration pipelines that integrate Mythril and Slither, and possibly EthRacer to perform tests.
- In addition, we recommend that developers with the available resources and who are deploying smart contracts expected to receive high volume use the EthRacer package to protect against more difficult-to-detect, dangerous event-ordering bugs. If physical hardware presents an issue, we recommend that the developer considers using cloud services such as Amazon Web Services (AWS) or Microsoft Azure to fit the needed memory and computational requirements.
- If resources do present a constraint for using EthRacer, then we recommend that developers use Oyente to get at least minimal bug detection for transaction-ordering bugs. If this route is taken, developers should thoroughly understand event-ordering bugs and be very mindful in avoiding them when coding.
- Where possible, use libraries and code snippets that have been used and tested by other reliable sources (like `ReentrancyGuard`) to mitigate known attacks.

References

[1] BANSAL, K., KOSKINEN, E., AND TRIPP, O. Automatic generation of precise and useful commutativity conditions. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2018), Springer, pp. 115–132.

[2] CONSENSYS. Consensys/mythril: Security analysis tool for evm bytecode. supports smart contracts built for ethereum, hedera, quorum, vechain, roostock, tron and other evm-compatible blockchains. `https://github.com/ConsenSys/mythril`.

[3] DEL CASTILLO, M. The dao attack. `https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft/`, 2016.

[4] ETHEREUM. Ethereum virtual machine (evm). `https://ethereum.org/en/developers/docs/evm/`.

[5] ETHEREUM. Solidity is a statically-typed curly-braces programming language designed for developing smart contracts that run on ethereum. `https://soliditylang.org/`.

[6] ETHEREUM/REMIX. Remix ide: Deploy and run transactions in the blockchain. `https://remix.ethereum.org/`.

[7] FEIST, J., GRIECO, G., AND GROCE, A. Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)* (2019), pp. 8–15.

[8] GHALEB, A., AND PATTABIRAMAN, K. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. ISSTA 2020, Association for Computing Machinery, p. 415–427.

[9] KOLLURI, A., NIKOLIC, I., SERGEY, I., HOBOR, A., AND SAXENA, P. *Exploiting the Laws of Order in Smart Contracts*. Association for Computing Machinery, New York, NY, USA, 2019, p. 363–373.

[10] LUU, L., CHU, D.-H., OLICKEL, H., SAXENA, P., AND HOBOR, A. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, Association for Computing Machinery, p. 254–269.

[11] SWENDE, M. H. Transaction ordering: Some thoughts about how a node can order transactions in a block. `https://notes.ethereum.org/@holiman/H17hFNWfd`, 2021.

[12] TIKHOMIROV, S., VOSKRESENSKAYA, E., IVANITSKIY, I., TAKHAVIEV, R., MARCHENKO, E., AND ALEXANDROV, Y. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends*

in Software Engineering for Blockchain (New York, NY, USA, 2018), WETSEB '18, Association for Computing Machinery, p. 9–16.

[13] TSANKOV, P., DAN, A., DRACHSLER-COHEN, D., GERVAIS, A., BÜNZLI, F., AND VECHEV, M. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2018), CCS '18, Association for Computing Machinery, p. 67–82.

[14] YU, G., ZHAO, S., ZHANG, C., PENG, Z., NI, Y., AND HAN, X. Code is the (f)law: Demystifying and mitigating blockchain inconsistency attacks caused by software bugs. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications* (2021), pp. 1–10.

[15] ZHANG, P., XIAO, F., AND LUO, X. A framework and dataset for bugs in ethereum smart contracts. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2020), pp. 139–150.