A Case Study of Security Vulnerabilities in Smart Contracts

Marvin James Rautenberg, Filip Rezabek*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: marvin.rautenberg@tum.de, rezabek@net.in.tum.de

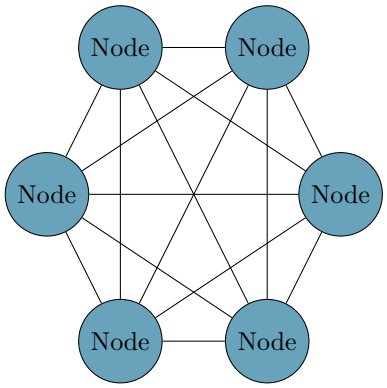
***Abstract*—Ethereum is the first blockchain network that introduced smart contracts which is code that can be executed on a distributed and publicly visible ledger. This makes a trustless and secure system of transaction possible that can not be altered after execution. As a result handling transactions and contracts is significantly improved no matter if the data being processed is tangible or intangible. To ensure this system is appropriate for use in a large scale it is important to analyze the security of it, what possible vulnerabilities the programming language has and how to minimize them which we conclude in a case study that refers to related work and combines all the conclusions. Subsequently we come to the deduction that Turing Completeness is rarely needed in terms of functionality in smart contract programming languages and rather harms the security of it.**

***Index Terms*—ethereum, blockchain, smart contracts, solidity, turing complete**

1. Introduction

Blockchain Technology is steadily growing in popularity and importance posing as one of the most interesting new asset classes on the finance market as even banks now invest into blockchain technology like Bitcoin and Ethereum. Whereas Bitcoin is very limited, Ethereum expanded greatly in the number of use cases it has by introducing the first version of smart contracts that make it possible to run code on a distributed network. This system can eliminate the need for trust in sensitive areas like financial transactions which make transactions much more automated, secure and stable. Having no need for a middle-man to conduct the transaction makes the blockchain a tool to verify and track every transaction that is made on the network.

Even though smart contracts bring multiple advantages compared to traditional contracts, there is the question of how secure this new system is and how it relates to Turing completeness. Additionally the security of the programming language and how to improve the security of the language specifically is important. The paper explains the key concepts needed to understand the analysis in Section 2, then analyze the design of smart contracts and find the connection to security in Section 3. After that we conduct a case study on where the sources of vulnerabilities in the smart contract language Solidity are and how to reduce them in Section 4 and talk about related work in Section 5. Following we come to the conclusion why Turing completeness is rather counterproductive in respect to security in Section 6.



[3]
Figure 1: Blockchain Network

2. Background

2.1. Blockchain

A blockchain is a distributed electronic ledger which records transactions and tracks assets [1]. It is crucial for applications where traditionally you need a trusted middleman to complete sensitive transactions. These transactions can range from a simple currency transaction to law documents and more since a blockchain can track tangible assets like houses or cars, but also intangible assets like intellectual property [1].

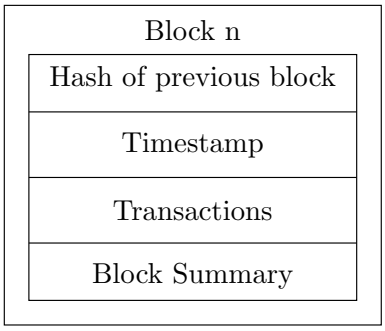
Distributed means it is a decentralized network of nodes like shown in Figure 1. These nodes can be computers running the software of a specific blockchain where every node is connected to each other instead of having a centralized hub of operations like a single server [2].

This means that all the data of the blockchain is publicly visible but the blocks containing the data are not modifiable. Blocks contain the hash of the previous block and multiple transactions in addition to other data like shown in Figure 2.

As this paper will largely focus on Ethereum's implementation of smart contracts we will look further into the attributes of the Ethereum blockchain as other blockchains with the option to create smart contracts are very similar to the Ethereum system.

Transactions on the Ethereum blockchain can only occur between an externally owned address (EOA) and another EOA, between an EOA and a smart contract, or between two smart contracts. An externally owned address usually represents a human made address also called wallet that has a private and public key. A public key is needed to be able to address a specific wallet for a simple

doi: 10.2313/NET-2022-11-1_10



[3]

Figure 2: Block Artchitecture

transaction and the private key is to sign transactions. The private key is used like a PIN on a credit card to confirm a transaction before it is executed. Whereas an EOA stores the balance and nonce which counts the number of all confirmed transactions, a contract-address additionally has code and a storage to track the states it has. This means that a contract address has at least the same amount of actions it can execute as a human created address.

Transactions can not be altered after execution which eliminates the need for trust in a transactions. Usually you have a system in place that needs to be trusted to make sure transactions take place as intended like a bank making sure they deliver your payment and in turn make sure that the seller in a transaction receives his payment from the buyer. Both the seller and the buyer expect the bank to handle everything related to processing the payment and thus have to trust it. The system depends on this trust which is a disadvantage as the trust can be exploited by malicious bankers and the way of making transactions would fall apart if banks became untrustworthy. Smart contracts on the other hand eliminate what would be the bank in a transaction and introduce an electronic contract that is executed on all nodes of the blockchain so everybody can verify that the contract is being executed correctly.

Only one node, the one which completes the contract the fastest actually alters the blockchain by adding the data to the next block of transactions that is chained to the rest of the blocks which in turn permanently alters the whole blockchain for all participants by adding a new block. This process is also called mining. For using the node's computational resources the blockchain rewards the node with the currency the blockchain uses which is Ether in the case of Ethereum. Being able to see all transactions and not being able to change transactions makes the blockchain "highly trustworthy, transparent, and incorruptible" [2]. For all the nodes to agree on certain data values to determine what is a correct output of a transaction there are consensus mechanisms like Proof of Work, Proof of Stake, Proof of Authority and more.

2.2. Smart Contracts

A smart contract is code that is executed in the virtual machine of a blockchain which in our case is the Ethereum Virtual Machine (EVM). The programming language of Ethereum is Solidity which is a Turing Complete programming language. Turing Completeness in the case of Solidity means it can run all programs a Turing Machine can run which mostly differentiates itself from Turing incomplete languages by being able to have complex programs including loops and recursion. The EVM works as a state machine that has an existing state, takes a transaction as an input and combines the two to create a new state. So the blockchain mostly consists of states that are changed over time. A Turing Machine and thus also a turing complete language have the important ability of disregarding the limitations of finite memory which will be an important fact in the analysis in section 3.

3. Smart Contracts Design and Analysis

3.1. Execution of Smart Contracts

You can use an Ethereum Node API to read data from the blockchain. Writing data to the blockchain is much more complicated than reading. You need to send a transaction to the Ethereum Network that specifies which smart contract to alter, which function to execute, any arguments you want to include, and if you are sending any Ether. After signing this transaction it needs a node to accept this transaction which does not have to also execute it. It is possible that the transaction will be forwarded to another node for execution. The transaction will be added to a transaction pool which will be only executed when there are enough transactions to fill a new block. To validate the transaction the transaction is passed as an input to the smart contract which is executed in the EVM. This transaction pool is handled by the miners that all simultaneously execute the transactions in the pool by solving a mathematical problem until one miner finishes all the transactions in the pool and solves the mathematical problem. So only one miner modifies the blockchain and adds a new block and all the other miners are used for verification of the result of the transactions but ultimately discard their calculations. Most of the other blockchains have a similar way of executing smart contracts.

Executing a smart contract is synonymous to buying something from a vending machine [4]. You enter at least the amount of money you need to buy a specific product, you press a button, you get your product and possibly some change back.

Since Solidity, the language Ethereum smart contracts are based on, is Turing complete you can not predict whether a program will finish or not which is why the developers of Ethereum have introduced gas fees to implement some of the benefits of having a Turing incomplete language. Gas fees are payed with every transaction. The person or contract trying to send the transaction has to specify how much they are willing to pay in gas for the transaction to arrive. It is possible to set the limit too low for the transaction to be declined in which case you will still lose the gas fees and the transaction will not be executed. If the gas limit is set appropriately to where the fees do not exceed the limit the transaction will be executed and possible remaining gas that is left over will be reimbursed to the sender of the transaction. Gas fees depend on how busy the Ethereum network is and change over time. Gas fees fix the disadvantage of Turing completeness disregarding the limits of finite memory which could lead to endless looping programs

doi: 10.2313/NET-2022-11-1_10

that never finish as the gas fees will at some point in the execution run out and stop the process. Blocks have an upper gas limit which can not be exceeded by the cumulative sum of the gas fees of the transactions that are in the pool of transactions for the specific block [2]. This ensures that not all transactions will be written into the same block. The nodes of the network can act as miners or EVM depending on the situation [2].

3.2. Other Smart Contract Languages

While Ethereum is the biggest blockchain with the ability to write smart contracts there are several alternatives like Algorand with the programming language TEAL, Cardano which is a blockchain platform using Proof of Stake and EOS.IO which is also a platform built for smart contracts. Algorand tries to solve the problem of scalability, speed of transaction and security that is common among blockchain technology networks [5]. The Ethereum Network can only handle up to 15 Transactions per second [5] whereas Algorand can process up to 1000 transactions per second [6].

The dramatic difference in efficiency is mostly due to Algorand using a different type of consensus mechanism. Reviewing scalablity between different platforms shows a trend of higher transactions per second often being accompanied by weaker security as higher security often implies more resource intensive concepts [5]. Ethereum uses Proof of Work as of the time writing this paper and Algorand uses Proof of Stake which does not require nearly as much resources and scales much better. A switch to Proof of Stake for the Ethereum Network is planned for the future. Algorand's Proof of Stake Mechanism uses the Verifiable Random function to randomly select a Holder of the Algorand currency to validate the next block in the chain instead of using miners like in the Proof of Work approach of Ethereum. A minimum amount of ALGO, the token of the Algorand network, needs to be pledged by a node to be able to validate blocks to ensure that the validator does not act maliciously as it would be unprofitable.

The security advantages TEAL has over Solidity come from it being a Turing incomplete language. Although Solidity is not as limited in the variety of algorithms it can compute like TEAL, being Turing incomplete is the key to reducing the possible attack vectors of the programs that are written with it. [7] concluded that at most 35.36% of smart contracts written in Solidity require Turing completeness to be executable. Therefore the majority does not require it and [7] states that it makes sense to rather use a Turing incomplete language. Although it is also mentioned that a mix of both language types could be the best option to still retain some of the more complex algorithm capabilities of Turing complete languages.

3.3. Security

Since Solidity is the first practical smart contract capable language [8] it does have the benefit of being most popular choice among smart contract developers [5] which makes identifying security vulnerabilities and adopting generalized good practices for coding in Solidity much easier than lesser known languages like TEAL. Major hacks like the DAO hack on the Ethereum network were only possible because of Solidity's Turing completeness, allowing reentrancy attacks causing major financial loss [9]. The EVM was working as intended and the DAO contract itself did not have any flaws but the language itself has flaws due to it being Turing complete which the designers of the language might have overlooked [9].

Most security vulnerabilities are flaws in the coding of the smart contract itself. Some of the most common vulnerabilities are reentrancy attacks and the use of oracle manipulation. Oracles provide data from outside the blockchain. An Oracle could be a sensor on a car tire to monitor the health of the tire to monitor if it is about to break. This kind of information is not on the blockchain as it is real-world data capture by sensors and oracles provide the connection needed to use this external data in smart contracts. Now depending on how sensitive the contract is the choice of oracle can be crucial for the smart contract to work as intended. If an oracle only gets data from one sensor in our example of the car tire, the sensor could be faulty and send wrong data which might trigger a chain of transaction that leads to an emergency call saying a tire broke even though it did not. Using multiple sensors would be needed to make the contract more robust. In other scenarios it is advised to use decentralized points of data sources in an oracle to make sure the data is correct and confirmed by many other sources as this is very important for the contract. Now this also presents an attack vector if you manipulate the oracle you can directly influence the execution of the transactions made by the smart contracts relying on this oracle. This is less a vulnerability of Solidity itself but rather a possible flaw in developing smart contracts made by the developers.

An example for a reentrancy attack could be two people writing each other letters, person A receives a letter from person B, starts writing an answer to the letter from B but does not complete it and starts writing a new letter concerning a different topic and sends it to person B. Now person B answers the letter sent by person A and A finishes writing his first response and sends it to B which would confuse B as it refers to a different conversation that was had before. This type of concept was used in the DAO hack to request Ether multiple times from a smart contract before the contract checked the balance which resulted in the attacker receiving more Ether than intended [10]. Attacks like these can be prevented by better coding practices discussed in section 4.

4. Case Study

4.1. Sources of vulnerabilities

Table 1 shows multiple known vulnerabilities of the Ethereum Network and on what level they appear on according to [11]. We can see that most vulnerabilities can be traced back to Solidity. Considering that the benefits of Turing completeness are not used most of the time in smart contracts made with Solidity it is reasonable to think that the smart contracts would be much more secure if the programming language used was not Turing complete without having to sacrifice too much functionality as the additional functionality of Turing completeness is rarely

TABLE 1: Security Vulnerabilities

	Solidity	EVM	Blockchain
Reentrancy	✓		
Type casts	✓		
Generating Randomness			✓
Gasless send	✓		
Immutable Bugs		✓	
Keeping Screts	✓		
Stack size limit		✓	
Unpredictable state			✓
Call to unknown	✓		

needed. Although it would be much more secure to use a Turing incomplete language we can not neglect the less than 35.36% that [7] concluded to be needing turing completeness. The quantity of the contracts using this complexity does not directly tell the importance of these smart contracts in the network. This 35.36% could be relied on by a lot of other smart contracts that do not need Turing completeness themselves so the influence on the network might be and is probably much higher than the aforementioned 35.36%.

It is important to carefully weigh the benefits of more security versus more functionality and decide which approach is more important in the application of smart contracts to be able to decide if the programming language should be Turing complete. Finding a way to combine both types of languages by having two separate languages that are Turing complete and incomplete to find a middle between the benefits and disadvantages as mentioned in section 3.2 seems to be the best option at the moment.

4.2. Guidelines for writing secure smart contracts

Despite the clear vulnerabilities of a Turing complete language like Solidity, it is possible to minimize the potential security issues in a smart contract by following coding principles. Auditing a smart contract depending on how important it is a good way of reducing the risks of security attacks. We can use static analysis like Slither which is a python program that would directly identify some of the biggest vulnerabilities and warn the developer if his code is prone for issues like Reentrancy. To create safe smart contracts we can not completely rely just on static analysis and need to use manual analysis tools like symbolic execution tools. Echidna is a symbolic execution tool where you can simulate transaction execution without running the code on the public blockchain. This allows us to use Fuzz-Testing to manually assess if functions work as intended. Vulnerabilities like generating randomness where the random values are not as random as the developer wants it to be can just be tested by creating a lot of values with the Fuzz-Testing tool and checking if values are repeated.

5. Related Work

Even though there is literature on similar topics like [12] and [11], they usually only focus on security aspects of smart contracts without connecting the vulnerabilities to the Turing completeness of the language and how this loss or gain in security weighs compared to the functionality. There is literature about the need of Turing completeness in smart contract programming but these mostly are in regard to functionality and do not make a connection to security as well. There is a lot of work regarding blockchain technology, smart contracts in general, how to write smart contracts and most of them refer to Solidity as it is one of the most commonly used languages for smart contracts. There are generally a lot of unscientific guides on how smart contracts work and what practices conclude in a more secure smart contract and what to avoid when programming with Solidity for example. Big attacks on the security of smart contracts are well documented like the DAO hack. This paper rather combines all of these findings and forms a new conclusion.

6. Conclusion

It seems using a Turing complete language has a large negative effect on security as most vulnerabilities can be linked to attributes that only occur in Turing complete languages like a program not terminating by itself. Problems like this can be reduced partly by introducing limitations that are more similar to Turing incomplete languages seen in the introduction of gas fees in the Ethereum network to combat the problem of a program not terminating by itself and thus wasting resources on the network. But it is not always possible to neglect the functionality of Turing complete languages which can provide crucial algorithm support that Turing incomplete languages do not. So either the combination of Turing completeness and incompleteness or using a Turing complete language and using strict security guidelines while creating smart contracts can be viable compromises to ensure security.

Generally the Ethereum Network seems like a very resource intensive network with the use of Proof of Work as the consensus mechanism and the Turing completeness of Solidity which allows for more wasteful and inefficient algorithms compared to a Turing incomplete language like TEAL which also works with a Proof of Stake consensus mechanism that allows the whole Algorand Network to be much more energy efficient than Ethereum. This not only reduces transaction times but also improves scalability and security which seems like an overall improvement.

References

[1] "What is blockchain technology? - ibm blockchain," 2022. [Online]. Available: https://www.ibm.com/topics/what-is-blockchain

[2] R. Modi, *Solidity Programming Essentials: A beginner's guide to build smart contracts for Ethereum and Blockchain*. Packt, 2018.

[3] T. Salman, R. Jain, and L. Gupta, "Probabilistic blockchains: A blockchain paradigm for collaborative decision-making," *2018 9th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pp. 457–465, 2018.

[4] R. Wilkens and R. Falk, *Smart Contracts: Grundlagen, Anwendungsfelder und rechtliche Aspekte*, ser. essentials. Springer Fachmedien Wiesbaden, 2019. [Online]. Available: https://books.google.de/books?id=k9UyyQEACAAJ

[5] G. A. Tsihrintzis and M. Virvou, "Advances in core computer science-based technologies," Jan 1970. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-41196-1_1

[6] N. Borisov and C. Diaz, *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part II*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2021. [Online]. Available: https://books.google.de/books?id=TuVJEAAAQBAJ

[7] M. Jansen, F. Hdhili, R. Gouiaa, and Z. Qasem, "Do smart contract languages need to be turing complete?" *Advances in Intelligent Systems and Computing*, p. 19–26, 2019.

[8] D. Gerard, C. Wagner, K. Boyd, and B. Gutzler, *Attack of the 50 Foot Blockchain: Bitcoin, Blockchain, Ethereum & Smart Contracts*. David Gerard, 2017. [Online]. Available: https://books.google.de/books?id=R7hEDwAAQBAJ

[9] H. HackerNoon, "Should smart contracts be non-turing complete?" Jul 2019. [Online]. Available: https://hackernoon.com/should-smart-contracts-be-non-turing-complete-fe304203a49e

[10] M. Derka, "What is a re-entrancy attack?" Aug 2019. [Online]. Available: https://quantstamp.com/blog/what-is-a-re-entrancy-attack

[11] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts," *https://eprint.iacr.org/*, 2016. [Online]. Available: https://eprint.iacr.org/2016/1007.pdf

[12] N. F. Samreen and M. H. Alalfi, "A survey of security vulnerabilities in ethereum smart contracts," *ArXiv*, vol. abs/2105.06974, 2021.

doi: 10.2313/NET-2022-11-1_10