HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELOR THESIS

Herr
Jonas Bentke

Analysis and realisation of acceptance criteria for pending transactions in an ethereum-blockchain environment

2017

Faculty of **Angewandte Computer- und Biowissenschaften**

BACHELOR THESIS

Analysis and realisation of acceptance criteria for pending transactions in an ethereum-blockchain environment

Author:
Jonas Bentke

Study Programme:
Angewandte Informatik

Seminar Group:
IF14wS-B

First Referee:
Prof. Dr.-Ing. Andreas Ittner

Second Referee:
Msc Steffen Kux

Mittweida, September 2017

Bibliographic Information

Bentke, Jonas: Analysis and realisation of acceptance criteria for pending transactions in an ethereum-blockchain environment, 43 pages, 11 figures, Hochschule Mittweida, University of Applied Sciences, Faculty of Angewandte Computer- und Biowissenschaften

Bachelor Thesis, 2017

Abstract

The Blockchain is a technology which has the capabilities to change the way, the world operates. As promising as this may be, there are still many challenges which do not exist or are way simpler to solve in conventional software solutions. Services which are offered over the blockchain suffer from so called Block-confirmation-times where the customer simply has to wait till the transaction is confirmed. In this paper possible solutions to that problem will be examined and challenges that arise from the specific criteria of the Ethereum Blockchain will be analyzed.

I. Contents

II. List of Figures

1 Motivation

At the very least after the dramatic price increase of cryptocurrencies, even non technical people have probably heard of blockchain by now. With its unique ability to create a trust less network of nodes, crypto currencies are taking the financial world by storm with a market cap of more than 145 billion USD at the time of writing [06]. As many new cryptocurrencies are emerging from this futile ground, everyone from startups to even banks and big corporations start to take a big interest in this enabling technology [02].

The driving force behind crypto currencies is the blockchain, a cryptographically secured ledger that is shared by everyone on the network. It consists of blocks, chained together with the hash value of the previous block, thus making the whole chain temper proof. First introduced by Satoshi Nakamotos whitepaper about Bitcoin [17], many different Blockchains emerged into the space.

The Ethereum Blockchain is even in comparison with the Bitcoin Blockchain still in its infancies. Running live since late July 2015 [10], it developed into a rapidly growing cryptocurrency that attracts developers with its own programmable virtual machine. The ability to code so called smart contracts creates endless possibilities to create and control value in a digital form.

Thanks to that rapid growth, more and more online retailers start to accept Ethereum's own cryptocurrency Ether, next to Bitcoin. Customers can pay without a middle man and do so anonymous and without having to trust a third party to handle their money. That not only saves money for the customer but also for the vendor.

Nevertheless, the blockchain technology comes with its own challenges. One of those problems is the topic of this paper. Sending and receiving Ethereum transactions is like in many Blockchains not instant and needs to be taken into consideration when dealing with customers. While a customer expects that payment is instant, neither Bitcoin or Ethereum can provide that service. Two major reasons for that issue are clearly at the forefront of this topic: Long blocktimes and uncertainty about the current view of the blockchain.

Long blocktimes mean that transactions send to the network need at least that amount of time to be accepted as valid. That means currently around ten minutes for Bitcoin and twenty-two seconds for Ethereum. To ensure that the transaction is valid and cannot be changed again, the user has to wait for more blocks to confirm his transaction [21].

The goal of this work is to enable vendors to accept pending transactions with a certain level of security, meaning to solve the issue of long blocktimes and network uncertainty. This is done by implementing filters that check all incoming transactions for a set of

criteria given by the vendor.

Taking the age of Bitcoin, research on the topic is widespread and available [21] [14] [03] [16]. Ethereum on the other hand is still lacking much of the research and attention that is given to Bitcoin. This thesis gives an introductory entrance into the topic and explores the already done research on the Bitcoin Blockchain and tries to apply those to the Ethereum Blockchain. Following a basic explanation of the blockchain technology and specific properties of Ethereum, the actual problem of accepting pending transactions is discussed. Many solutions of the Bitcoin research is applied and introduced. To conclude this thesis, an application is introduced that at least solves the issues under certain conditions.

2 Blockchain

This chapter explains the basic Blockchain-technology. It reflects on the roots and development of blockchain and analyzes the general concept. Following, the Ethereum Blockchain is examined and analyzed.

2.1 Historical Background

Ever since the Internet began, science has been dealing with a crucial topic. How can you ensure that a digital document of any kind is the original? Stuart Haber and W. Scott Stornetta have already examined in "How to Time-Stamp a Digital Document" [13] this problem in 1991. The motivation for this paper was the problem of proving the ownership of a digital photography [01]. They reported on two possible methods to sign documents[1] with a timestamp.

The first method is a central approach which transmits the hash of a document to a timestamping server. The hash will then be signed together with a timestamp, an ID of the sender and the hash of the previously signed document. The signed message will then be transmitted back to the sender. As soon as someone doubts the ownership of such a document, the owner can prove through the hash of his signed document and the hash of the following documents that he was at least the first one to timestamp his document. For someone to claim ownership for that document, he would have to provide a signed document which contains an earlier timestamp.

For the second method Haber and Stornetta introduced a decentralized solution. The client would send the hash of the document and his ID to k random participants of the network. Everyone, who would receive the message, includes a timestamp, signs it and sends it back. The proof of ownership would hereby come through the list of signed messages. Assuming that the majority of the network is trustworthy, the chance of sending the message to k/2+1 ill intendant nodes of the network is vanishingly low. This method was later called the "Random Witness Protocol" [04].

One year later Haber and Stornetta published together with Dave Bayer the improved versions of their concepts [04]. They proposed the idea "to merge many unnoteworthy time-stamping events into one noteworthy event" [04, S. 2]. Using tree structures (today called merkle trees) they brought down the number of verification steps significantly. They achieved that through hashing the hash values of two documents together and building a tree structure upwards. That way only one hash has to be verified for an

[1] "Of course, digital time-stamping is not limited to text documents. Any string of bits can be timestamped, including digital audio recordings, photographs, and full-motion videos." [13, p. 10]

endless number of documents.

Following their improvements they unified their three different approaches to build a system that could be seen as the origin of blockchain:

> "...we imagine that the three methods may be used in a complementary fashion, as the following example illustrates. An individual or company might use linear linking to time-stamp its own accounting records, sending the final summary value for a given time period to a service maintained by a group of individuals or parties. This service constructs linked trees at regular intervals. The root of each tree is then certified as a widely viewed event by using the random-witness protocol among the participants." [04, p. 6].

With the help of such a system can be proven who was the first to timestamp a document. Effectively that means the authorship of the document will be verifiable. The system cannot however verify the ownership of such signed document. This might not be an issue for general documents or even pictures, given that in those cases an authorship is enough to collect fees and such, but it is definitely not enough for digital values.

The previously examined algorithms do not intend to provide a way to move the authorship from one person to another. The following example illustrates that case: Alice and Bob make a business transaction in which Alice promises Bob the authorship of a digital document. Alice already signed the document with a timestamp using one or all of the previously examined algorithms. In order to transfer the authorship to Bob, Bob would need to sign the document with his ID using the same algorithm as Alice. The issue that arises now is that Bob has no way to prove that he has acquired the document from Alice in a legal and honest fashion. To avoid that, Alice includes in her data a message that states that she willingly transferred authorship to Bob. If a third person questions Bobs authorship (which became now ownership), Bob can simply provide the plaintext message. If the Hashes match, Bob proved his ownership.

Problems will arise as soon as Bob wants to confer the ownership to Carol. Bob and Carol follow the same algorithm as Alice and Bob. Carol can now prove that she received the ownership from Bob, and that Bob received his from Alice. The real issue arises now if Bob decides to confer his ownership not only to Carol but also to Dave. This is commonly known as a "Double Spending Attack"[2]. Dave can now prove that he has the ownership as well as Carol. To avoid such a predicament, an algorithm is needed which provides every participants with the same basis of information at all times.

Building upon the discoveries of Bayer at al, the whitepaper "Bitcoin: A Peer-to-Peer Electronic Cash System" by Satoshi Nakamoto [17] was published. Not long time after

[2] www.investopedia.com/terms/d/doublespending.asp

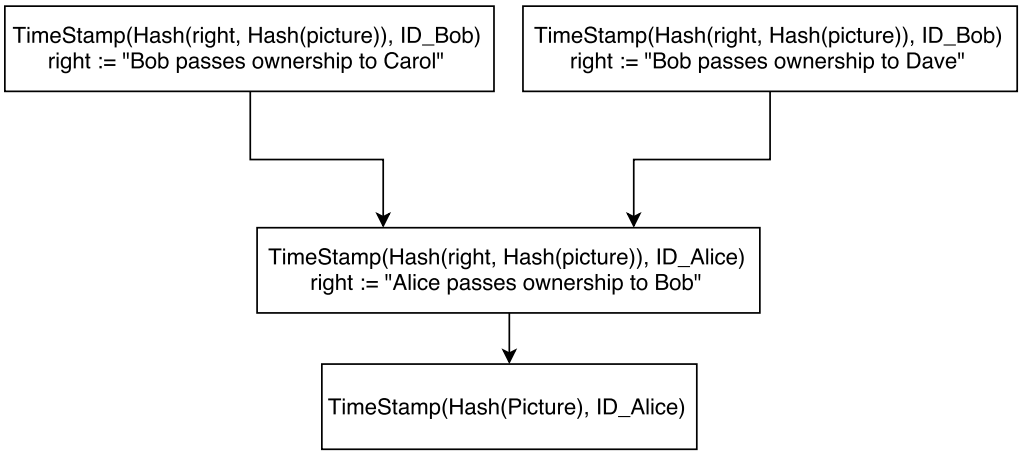

Figure 2.1: Example: Double Spending

followed the first implementation of Bitcoin as an open source project.

2.2 Blockchain in general

This chapter covers the basic structure of a blockchain the way it is described by the Bitcoin Whitepaper. It will not however explain Bitcoin specific implementations which are not necessary to understand the Ethereum Blockchain.

One block in the blockchain contains as a bare minimum a timestamp, one or more transactions and the hash from the previous block. This can prove that these transaction existed at the moment in time where the block was created [17, p. 2]. If an attacker tries to change a transaction in one block, he will unintentionally change all following blocks. Responsible is the hash of the changed block. If the hash of one block changes, then the next block, which contains the hash of the previous block, would not point to his original predecessor anymore and thus the chain would be broken. This construct is similar to the linked list described by Hebert and Stornetta [13, p. 5]. In order for this algorithm to work on a peer-to-peer network, there needs to be some way to create consensus between the participants of such network (peers).

Bitcoin delivers such an algorithm to ensure that the whole network has the same blockchain. The first step to ensure that everyone in the network knows, or can know, about each transaction which is send, is to propagate transactions throughout the whole network. Therefore each transaction in the bitcoin network is public. In order to create consensus about which transaction is going into the next block and is consequently valid for everyone, the Bitcoin Blockchain relies on the proof-of-work algorithm [07], which is at this moment the consensus algorithm for the Ethereum Blockchain as well . Proof-of-work functions in a way that in order to create a block, the so called miner needs to find a hash for that block, that matches a given difficulty. The difficulty is set by the number of zeros, which need to precede the hash. In the block itself is a value named nonce,

that can be changed by the miner in order to create a different hash. As soon as a valid block is found, it is propagated to the network.

Everyone that receives the block can check for its validity by creating the hash of the block. To give an incentive for miners to find new blocks and subsequently spend money on the electricity necessary, each block is rewarded with a certain amount of bitcoin.

Everyone in the network can create the next block. If two or more nodes are able to find the next block at the same time, the blockchain is split apart. Every node accepts the first block that arrives as valid and chains it at the end of their blockchain. When the other correct block arrives, it is rejected because it is in the eyes of the node not valid (it does not contain the hash of the new block). Because other parts of the network received the other block first, some work on the next block for the one, and some for the next block of the other blockchain. This goes on until one blockchain is longer than the other. Every member of the network is encouraged to accept the longest blockchain as valid. If the node realizes that it is on a shorter chain, it rejects it and moves to the longer one.

In order to change a transaction in the past, the attacker has as well to change all following blocks or create new ones, until he created a longer chain as the currently used one. The other honest miners however create blocks at the same time as the attacker, which creates a race condition for the longest chain. In practice, the attacker would need more than fifty percent of the computing power of the network.

2.3 Ethereum

Ethereum was developed as an alternative to Bitcoin and delivers a number of functionalities which are not or only under great constraint implementable in Bitcoin. This chapter examines how the general concept of blockchain is implemented into the Ethereum Blockchain and what features are available that are missing in the Bitcoin Blockchain.

2.3.1 Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) [05] is defined as a quasi[3] turing complete machine. It is a virtual state machine, which changes the state as soon as a command is executed. The architecture of the machine is simple stack based and can be seen in Figure 2.2, which shows a schematic overview.

There are two memory spaces in the virtual machine. The first is a volatile memory

[3] "...the quasi qualification comes from the fact that the computation is intrinsically bounded through a parameter, gas, which limits the total amount of computation done." [22]

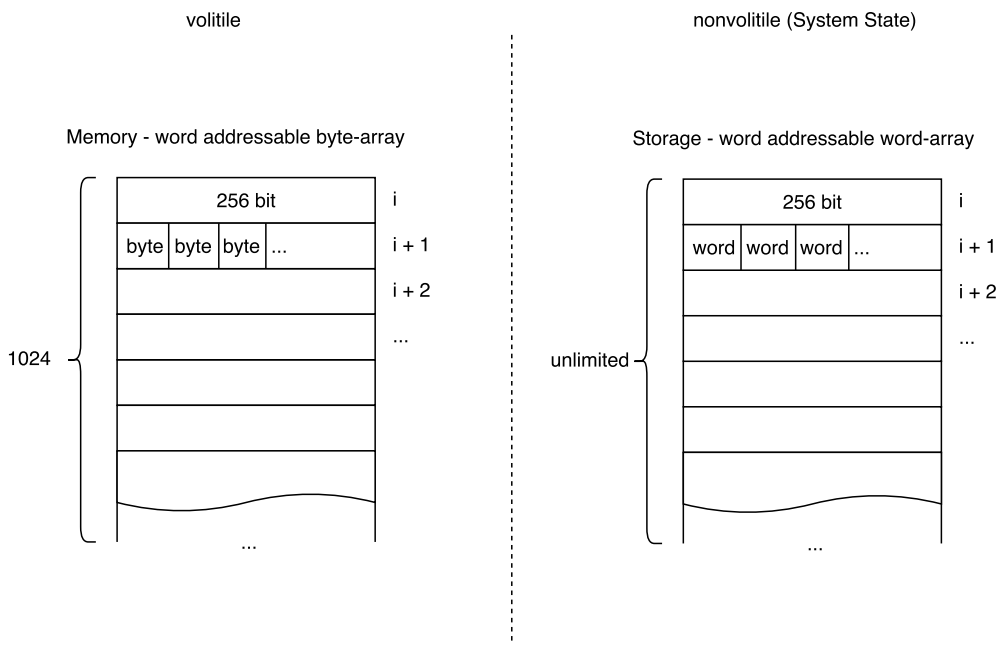

Figure 2.2: Overview of the memory spaces in the EVM

which is used for the execution of code. It is represented by a 1024 byte long array. It is addressed by a 256 bit word which leaves the individual memory cell at 256 bit.

The second memory space is non-volatile and contains the current state of the virtual machine. It is a word addressed word array and does not have a set length.

The code that is to be executed is stored in a read-only-memory.

A concept of two different types of addresses that is implemented in Ethereum emphasizes the advantage of such a virtual machine. An address is a 160 bit long asymmetric key pair. Ethereum separates between externally controlled addresses and contract addresses. Externally controlled addresses (accounts) are controlled by the private key, which usually represent users of the network. These accounts can hold Ether (the internal virtual currency of Ethereum) and sign transactions. Contract accounts (also called smart contracts) on the other hand are controlled by the contract code. They can hold ether as well, but do not have a private key and can therefore not sign transactions. The code that controls the contract is written into the state of the address with a special transaction that holds the contract code. Thus the contract code is immutable. Tasks for the code are send to the address with a message call and run by the virtual machine.

To execute code or change the state of the EVM the user has to pay a fee. "In order to avoid issues of network abuse and to side-step the inevitable questions stemming from Turing completeness, all programmable computation in Ethereum is subject to fees." [22]. Each opcode of the EVM is given a specific fee [22], which is taken by the

creator of the block[4]. That creates native protection against potential DoS attacks and general network abuse, but also to protect the user against several errors in his code, like endless loops etc. Each transaction is given a limit on how much it is allowed to spend. If the limit is reached, the transaction is stopped and all changes are reverted. All fees are stated as gas. Gas is an internal unit which is given a gas price by the sender of the transaction. It only exists within a transaction and can not be transferred or sold.

Gas is paid in Ether. Ether serves as payment for gas and as reward for generating a new block.

2.3.2 State

Ethereum is viewed as a complex state transition machine. Transactions which are contained in a valid block, change the state of the machine. The state is a mapping between addresses and address states. Each state combined creates the state of the blockchain. Every state consist of four values:

nonce number of transactions send from this address.
balance The amount of Wei[5] hold by this account.
storageRoot Merkle Patricia Tree [11].
codeHash The hash of the EVM code of this address. Is empty if it is not a contract address.

Figure 2.3 shows the mapping between the addresses and the states of the addresses.

The current state of the EVM is hold in the non-volatile memory area. As soon as a new block arrives at the node, all transactions from the block are executed to change the state of the virtual machine.

2.3.3 Transactions

To get the state of the current blockchain, all transactions in every block need to be executed in the right order[6]. Ethereum transactions are build in the following way:

nonce The nonce in a transaction is generally the same nonce as from the sender address. If the transaction is send with a higher nonce then that, it can only be

[4] The costs are calculated with the set gas costs for each opcode as described in the yellow paper and the gas price which is set by the sender of the transaction. The miner can decide if the gas price set by the user is high enough to cover his costs to take the transaction into his block
[5] Wei is the smalest unit of Ether in Ethereum. [22]
[6] The state is changed after each valid block. If someone joins the network at a later point in time, he calculates the current state in that way

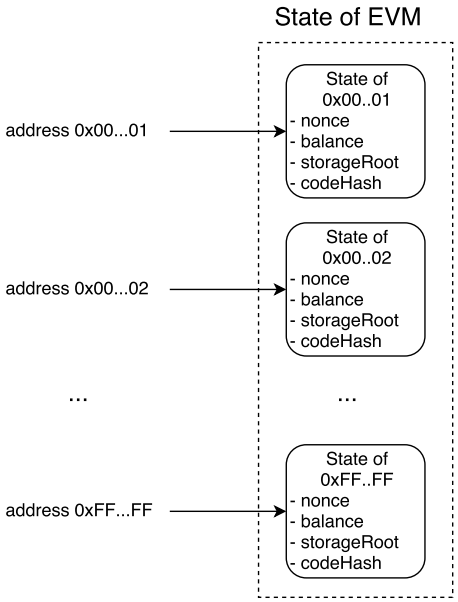

Figure 2.3: Mapping of addresses within the state

executed when the nonce of the address becomes the same value.

gasPrice The amount of Wei that is to be paid per unit of gas.

gasLimit The maximum value of the transaction. When the value is reached before computation is done, an error occurs. If an error occurs this amount of gas will be burnt and all state changes will be reverted.

to The address of the receiver.

value The amount of Wei that is to be sent with the transaction.

data A byte array that contains information about the code that it wants to call from a contract. If no contract is called, it is not used by the EVM. Every byte raises the cost of the transaction. By definition, the array is unlimited in size, but in practice limited by the block gas limit.

If the transaction is used to create a new contract, the structer of it changes the following way:

init Init is used instead of the data field. It contains a limitless byte array which contains the code of the transaction that is supposed to be created. It "is an EVM-code fragment; it returns the body, a second fragment of code that executes each time the account receives a message call (either through a transaction or due to the internal execution of code). init is executed only once at account creation and gets discarded immediately thereafter." [22]

2.3.4 Block

A block in the Ethereum Blockchain contains 3 parts: a blockheader, a list of transactions and a list of ommer[7].

The blockheader consists of a number of hash values, such as the hash of the previous block, the hash of the ommer list, the root of the transaction trie and the root of the trie of the transaction receipt. Transaction receipts are the result of a transaction. They contain the state before the transaction, the amount of used gas, logs and the bloom-filter for the logs. Next to those hash values are different meta-data such as the block difficulty, the block number, the gas limit, the actually used gas, a timestamp and a mixed hash which is used together with the nonce to create the hash for the consensus. Ommer, also known as uncles, are blocks that stand next to the previous block (have the same parent). Ommer are necessary to reward and think of the miner that created the correct block but did not get accepted fast enough by the network. The block difficulty uses these blocks to adjust the difficulty of the proof-of-work algorithm. Miners receive rewards for ommer.

[7] Gender-neutral term meaning "sibling of parent", see http://nonbinary.org/wiki/Gender_neutral_language#Family_Terms

3 Pending Transactions

After reviewing the most critical parts of the Ethereum Blockchain, it is important to note one of the problems that arises through the consensus itself. While it might not be as troublesome to wait for a new block to be generated when processing normal transactions, it definitely opens up problems for fast transactions[8].

Viewing a use case such as an ATM withdrawal, it might seem unacceptable for a customer to wait for potential 120 seconds[9] till the transaction is accepted into the blockchain. Given a more acceptable use case such as ordering a product online, the blocktime does not play a big role. The question arises when it is actually acceptable to wait and at what point it might even be necessary to accept pending transactions[10]. In order to discern what might be acceptable, the following chapters will discover possible attack vectors, different use case scenario and an implemented way for accepting pending transactions for payment, while minimizing the risk involved.

3.1 Transaction Acceptance

In order to know whether a transaction is ready to be accepted, it needs to be discussed how possible attack vectors might interfere with that. Given the shared base idea of blockchain between bitcoin and ethereum, there are some important differences in the actual implementation which need to be shown in more detail.

The most prominent group of attacks are widely called double spend attacks. While Satoshi Nakamoto claims in his whitepaper that this problem is widely solved in bitcoin, he also states that "the system is secure as long as honest nodes collectively control more power than any cooperating group of attacker nodes" [05]. While this is true for a double spend attack that relies on creating a hidden branch of the current blockchain, and publishing it after the vendor confirmed the transaction[11], it does not hold against a double spending attack that relies on a race condition. Karame et al expand on that attack in their paper "Two Bitcoins at the Price of One? Double-Spending Attacks on Fast Payments in Bitcoin" [14] and explain a scenario where the attacker sends a transaction

[8] Fast transactions refer here to transactions that most of the time involve face to face trading, where it is uncommon to have a long payment process opposed to paying for and receiving a certain good.

[9] The current Blocktime reaches 23 seconds on average, but can be as long as 120 seconds. It depends on when a miner finds the next valid block.

[10] A pending transaction described here, is a transaction that has been published to the network, but is not yet accepted into the next block. The reasons for that vary from being invalid to not having a lucrative gas price set.

[11] As proven by Meni Rosenfeld in his paper "Analysis of hashrate-based double-spending" [21], it is secure, given that only a certain percentage of hashpower is controlled by the attacker and the vendor waits n blocks for confirmation.

T to the vendor and sending a transaction T' to the miner. Given enough proximity to the vendor and potential help from attacker nodes, the transaction send to the vendor can become void after the next block. The Bitcoin client rejects any transaction that contradicts a previously collected transaction so that the attacker can send out T' with help of other attacker nodes faster to the network. If the vendor accepts pending transactions, the attacker can get away with the goods received, and T' will most likely be accepted into the next block annulling T.

Because Ethereum is build upon a blockchain as well, it has the same attack vector as just discussed for Bitcoin. Taken into account that the blocktime for Ethereum is much lower than for Bitcoin, it is more difficult but still possible to run such an attack. For fast transactions it is still required to have a nearly instant confirmation, e. g. in a supermarket where everyone else would have to wait in line, while the cashier is waiting for the block-confirmation.

While the bitcoin client simply rejects transactions that contained spend coins, Ethereum does not rely on unspent transaction outputs[12]. As explained in chapter 2.3.2 each address has its own state, which results in nodes simply relying on the previous state to check if a transaction is valid. This is done by simply checking the balance of the account at the previous state and checking if the nonce of the transaction is equal to the nonce of the address +1. Transaction with a higher nonce are stored for later and transaction with a lower nonce get rejected. Reviewing the two most commonly used clients for Ethereum, Geth and Parity, the way they implement the transaction pool for pending transactions shows that instead of rejecting incoming transactions with the same nonce, they check which transaction has a higher gas price. This is a critical step. In the previously explained attack vector, the attacker would just have to send a transaction with a higher gas price and would in that way overwrite the old transaction. This fact implies that the vendor can never be sure that a pending transaction is going to be executed or is being overwritten.

3.2 Acceptance of Pending Transactions

The time that is acceptable to wait for confirmation depends on two factors: the value of the transaction and the inconvenience for the customer. Figure 3.1 shows a matrix representing the correlation between inconvenience and time and which influence it has on pending transactions.

For high value pending transactions with no or only low inconvenience for the customer, it is a good practice to reject them. This might be the case at an online shop. Given that

[12] An unspent transaction output (UTXO) is a value of bitcoins that has not yet been assigned to be a transaction input. Bitcoin transactions are only valid, if the transaction output is backed up by enough UTXO. Following that logic, all Bitcoins can be traced back to the time they were mined.

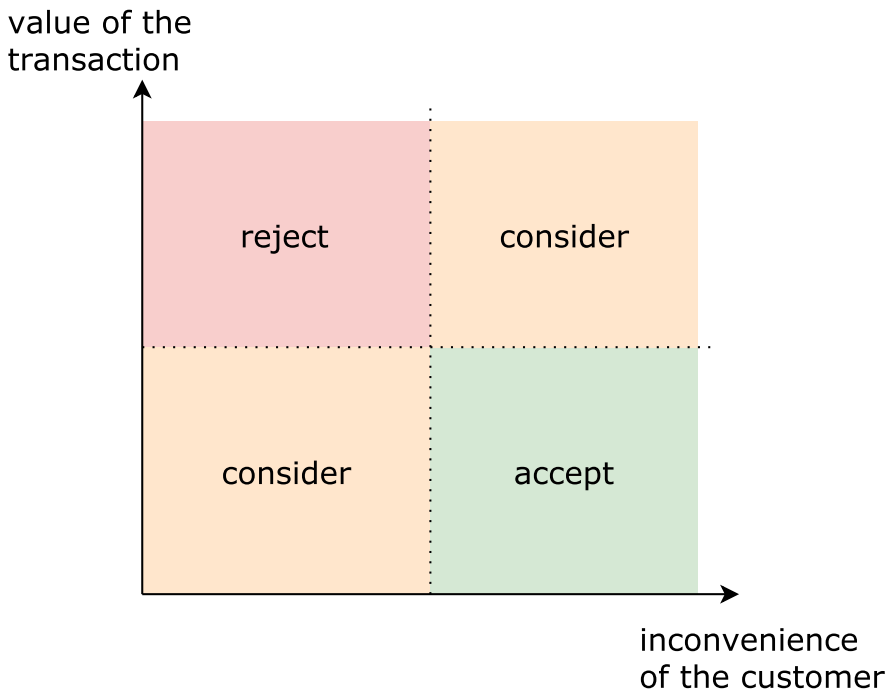

Figure 3.1: Representation of the correlation between customer inconvenience and value of the transaction.

the customer has to wait at least on day for the package anyway, the blocktime will not make much of a difference. Another example would be, after eating at a restaurant. In a space where the waitress will usually take your credit card into the back room anyway, it will not represent an inconvenience to wait for the next block to be confirmed. While it is an issue with blocktimes like bitcoins, it is not for the comparatively low time necessary to wait at the Ethereum Blockchain.

For transactions that have a low value and would create a big inconvenience for the customer, it is reasonable to accept pending transactions. The loss of the transaction would only be the value of the transaction itself. That cost is still comparatively low to the cost of a bad customer experience. In an article called "The Value of Customer Experience, Quantified" [15], Dr Peter Kriss shows that " . . . customers who had the best past experiences spend 140% more compared to those who had the poorest past experience.". Given that statement it is in the best interest of most companies, in regards to the low transaction value, to accept pending transactions for the sake of the customer. An example would be a low cost renting experiences like gaining access to wifi for a couple of minutes. In that example it would be unreasonable to wait for at least the next block when the actual usage only takes a couple of minutes.

The other two cases, low value and low inconvenience or high value and high inconvenience, depend on the use case. Looking at a low value and low inconvenience process, the tendency might be to wait for the next block. On the other hand, considering the use case of an fully automated system, it might not be practical to wait, if it would slow down

the whole process as such. Looking at a high value high inconvenience processes, the tendencies gos towards rejecting pending transactions. But again, there might be use cases where that is not necessarily the case, for example when it is not likely that the customer will run a double spending attack, like at a charity fundraiser event.

Taken into account these economic factors, the next section will give a technical perspective on the matter.

3.2.1 General Problems with Transaction-propagation

The main issue that needs to be looked at is the problem of the network topology. Given the fact that Ethereum is build on top of a peer-to-peer network, there is no guarantee that the vendor node will even receive the customer transaction. If the customer is broadcasting the transaction from a network position on the opposite site of the vendor, the next block might be mined before the transaction can even be pending at the vendors node [16]. In that instance the customer could be asked to send his transaction directly to the vendor node, which would open up the vendor for common double spending attacks[13]. Closing the incoming connections, "... essentially prevents the attacker from achieving one of the necessary conditions required to successfully perform a double-spend attack which is to directly connect to the targeted vendor." [20]. Remembering that it needs to be considered if it is even worth the risk to accept pending transactions.

Another problem arises when the transaction from the customer is actually pending in the transaction pool of the vendor. He is now able to check if the transaction is valid and start his service. As soon as the customer receives the service, he can propagate a transaction that has the same nonce as the transaction before. To ensure that the new transaction overtakes the old one, he might have helper nodes which propagate the same malicious transaction over the network. Each node will replace the old transaction, if the new transaction has a higher gas price. If the new block contains the malicious transaction, the vendor lost against the attacker. To defend himself against such an attack, Karame et al suggested two possible solutions to that problem.

First they recommended a listening period, where the vendor waits for a few seconds and listens for malicious transactions. But given that, the attacker could just wait until the period is over, they suggested another method, which is to deploy observers over the network, to warn the vendor node, in case there is a second transaction with the same nonce propagated. These observers[14] listen for transactions on the network and alert the vendor in case of a positive finding. Because they are widely spread over the

[13] Accepting transactions directly to the node, allows an attacker to start a double spend attack as described in chapter 3.1 but also opens the node up for an Eclipse attack [03]. The direct impact on known topology is explained by Matthias Lei in "Exploiting Bitcoins Topology for Double-spend Attacks" [16]

[14] There is an improved version of these observers described by Podolanko [20]. They are "âĂęa hybrid of observers and the peer alert system".

network, the chance that they will pick up such a transaction is in fact higher as if the vendor would just listen himself.

The short blocktime from ethereum makes it in this case not only harder for the double spend attacker but also harder for the defender of such an attack. Listening for a period of time for malicious transactions, either with observer or with the vendor node itself, might just take longer than waiting for the next block. To that effect must the attacker deploy his malicious transaction in the timespace, between the listening period and before the next block is mined. The same is valid for the vendor who has to listen for malicious transactions long enough to be certain there is no second transaction from the attacker and not to long, else the inconvenience level will be to high for all friendly customers or the next block will be mined anyway.

A simpler workaround for the double spending in the Ethereum Blockchain is to make use of smart contracts. Before the actual act of purchase, the customer deposits a certain amount of Ether into a smart contract. The vendor can now accept pending transactions from the customer without fearing a double spend attack. In case a transaction is rejected or is invalid, the vendor keeps the deposit. This moves the risk and responsibility from the vendor to the customer. The smart contract can be a simple multi-signature contract[15] which needs a trusted third party to decide in the case of a dispute. The big disadvantage of such a workaround is the fact that the customer would need to open a deposit with each vendor he plans on trading with.

3.2.2 Problems with State Changing Transactions

Each transaction in the Ethereum Blockchain changes the state of the EVM. Following it will be differentiated between two kinds of transactions:

autonomous transactions These transactions are independent from any action of other accounts[16]. To this category count transactions which only transfer ether from one address to another and the most simple function calls. Simple function calls are described here as calls that can be done independent of any other call to that contract (except *suicide*).

non-autonomous transactions Every transaction whose outcome can be altered by another transaction is a dependent transaction. That category includes most function calls in more complex smart contracts. One simple example would be that transaction A is depending on permissions given by transaction B. If transaction

[15] A Multi-signature contract, or short multisig, is a smart contract with functions that can only be executed when a set number of participants agree to do so. An example would be a savings account that can only move money when two of three participants agree upon it.

[16] Exempt from this rule are transactions send by the sender himself that would move funds from the account in a way, that the transaction in question does fail because of insufficient funds. These cases were discussed in detail in the previous chapters. In the following discussion we assume that the sender account has always sufficient funds.

A is taken into the block before transaction B, it will not have the necessary permissions and subsequently fails.

To accept autonomous transactions is relatively straightforward. As soon as such a transaction is received by the vendor node, the nonce will be checked. If the nonce is higher then the current nonce of that address plus one, than there are pending transactions which need to be executed beforehand. In the case that there are none, or that all transactions are already executed it is tested if the transaction is valid[17] and pass it on for execution. If there are transactions which are not yet received by the vendor node, the pending transaction is put on hold till they arrive. This process is shown in Figure 3.2.

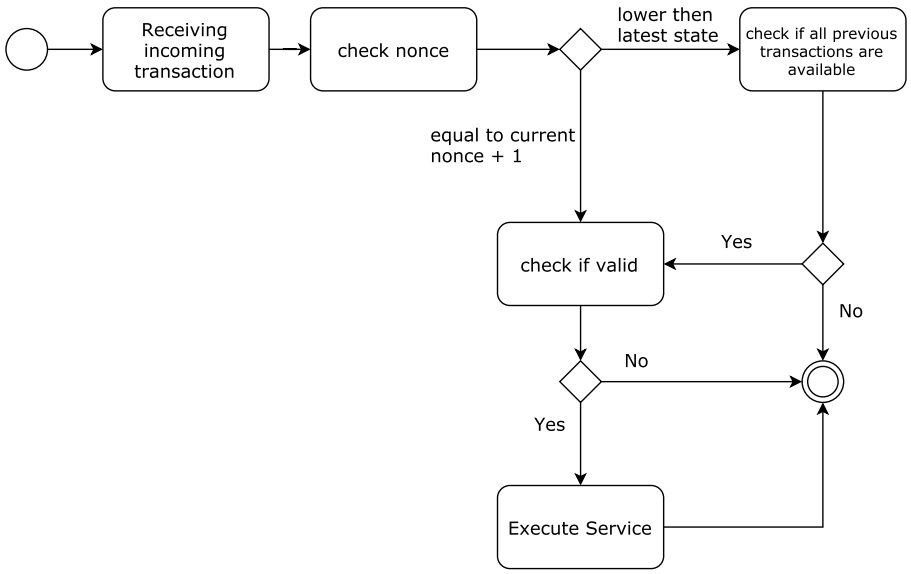

Figure 3.2: Flow chart of a simple transaction

Far more problems arise when trying to accept non-autonomous transactions. While it is not important for autonomous transactions to wait for other transactions to be processed beforehand, non-autonomous transactions suffer from the uncertainty that arises from the algorithm that decides the order of the transactions in a block.

When a new block is being put together for the blockchain, the creator of the block decides what transactions in which order will be executed. Looking at a simple example where 2 parties have to enter a value which will then be added together by either one of those parties. If the order in which the transactions are executed would be set, so that the final product will be calculated after the transactions of the two parties, everything will seem ok, but if the product calculation will be executed before one or both of the parties transactions, then the result will become incorrect (see Figure 3.3). Even knowing what strategy the miner might use to determine the order of transactions, makes it still uncertain, if he even has the necessary transactions in his transaction pool.

[17] The definition of a valid transaction is described in the Yellow Paper [22]

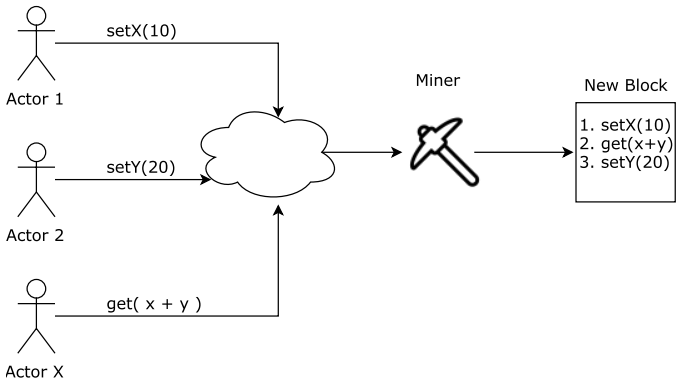

Figure 3.3: Transaction-ordering through miner.

Figure 3.4 shows a contract with two functions, an autonomous function *changeOwner(address owner)* and a non-autonomous function *set(uint paranumber)*. The autonomous function is changing the variable *owner* and can be called by anyone. Calling this function is possible, no matter what other transactions might do with the variable or the contract. In contrast, the non-autonomous function relies on the owner variable. Even if the user sends two transaction, *changeOwner* and *set*, in succession, it is not assured that the outcome of his first transaction will not be overwritten with a transaction from another account.

```
1  pragma solidity ^0.4.10;
2
3  contract Example{
4
5      uint public number;
6      address public owner;
7
8      function Example(){
9          owner = msg.sender;
10     }
11
12     function set(uint para_number) {
13         require(msg.sender != owner);
14         number = para_number;
15     }
16
17     function changeOwner(address newOwner) {
18         owner = newOwner;
19     }
20 }
21
```

Figure 3.4: Example contract for autonomous and non-autonomous transactions

This problem can be broken down into a single race condition. Between the moment where the transaction is send and the actual validation through the block, there is time for other transactions to change the state. Conventional programming languages like C deliver constructs like a mutex to prevent such behavior. Because there is no such thing in the EVM Opcodes, every transaction should be atomic in itself. Given, that this is not

alway possible[18], compromises are necessary.

3.3 Other Excluding Criteria

Blockchains present other environmental risks which can influence the criteria of accepting pending transactions. A major point is the chance of a fork. Forks happen when a split occurs in the blockchain itself. There are two kind of forks, a soft fork and a hard fork.

A soft fork happens when two valid blocks with the same parent block are propagated into the network. Some nodes will accept the one and some the other block. The protocol views always the longest chain as the valid one. Miners will now mine on the block they received until they find a new block. Every node will now jump on the chain that is the longest.

While a soft fork is happening on a regular basis, it provides some problems for accepting transactions in general. A vendor might accept a transaction and is then continuing on one particular fork. If his fork is the one that is going to be rolled back, all transactions since the beginning of the fork are invalid. This problem is not specific to pending transactions as such, except that systems for fork detection do not fully apply to pending transactions. The transaction between a customer and the vendor occur in that case before a fork can be discovered. As soon as the fork is detected, the transaction should be canceled or at least put on hold till the fork is resolved.

Hard forks are changes in the protocol itself and require an update of the clients. This kind of fork is not going to be resolved in the future but is permanent. Changes in the protocol include lengthy discussions between the core developers and are known to occur in advance. This will rarely affect pending transaction handling if the vendor prepares for that case. The only case where it could affect the vendor is if a change in the protocol would make the transaction invalid. Because both chains are constant after the fork, there is no loss in value for the vendor, as long as he runs a full client on his own.

Another financial risk is a DOS attack on the blockchain itself. If the vendor accepts a transaction but the network does not respond or all blocks are full, then he will only be paid as soon as the transaction is actually taken into a block. While a transaction is generally accepted or rejected relatively fast, in this case he could wait days or longer.

[18] Even if it is a simple accepting of a payment from a smart contract, there is no way to tell what will happen to the balance between providing the service and the block confirmation

4 Solutions - How to Accept Pending Transactions

A use-case analysis is performed and previous findings are applied. Criteria to accept pending transactions are created and taken into account while building an architecture for an implementation. A short analysis of the current implemented transaction pools provides a necessary insight into what kind of transactions need to be watched. A flow chart is shown that is implemented and explained at the end of the chapter.

4.1 Use-Case Analysis

The implementation of a possible solution is build around the following use case:

A customer wants to use, rent or buy a service from the vendor. The vendor is connected to the Ethereum Blockchain and runs his own node. In order to start the process, the customer sends a transaction with the necessary function call from his device to the Ethereum Blockchain. As soon as the transaction is propagated to the node of the vendor, it will verify the transaction. If it is valid, the process will be started.

There are several things to consider in that use case.

1. At what point is a transaction valid for the vendor?
2. What happens, when the transaction is not included into the next block?
3. What devices or things can be used, rented or sold with that system?

The validity of a transaction is defined in the yellow paper [22]. It checks if

- the transaction is well formed
- the transaction is signed properly
- the nonce is valid
- the gas limit is greater or equal to the intrinsic gas used
- the balance from the sender account is high enough to pay the upfront cost.

These rules of validity are implemented by the Ethereum client itself. In this use case, validity is seen in a wider scope. To accept a pending transaction, the vendor must be sure that this transaction will successfully execute. If the transaction is an autonomous transaction, a simple check against the current state is enough to accept it. That could be the case if the customer is buying a service. In most cases, the transaction will be non-autonomous, which requires additional checks for validation. The reason that most of those transactions will be non-autonomous is that the service generally supports only

a finite number of users. Because of that the transaction can fail, if another user's transaction is taken into the block faster. These additional checks defer from use case to use case. A possible solution could be to check the data portion of the transaction, provided the function call is known to the vendor. Analyzing all other pending transactions in the transaction pool can give a certain amount of security.

One option for the problem of transactions not being taken into the next block would be to stop the process immediately, but as shown in 3.2.3, the reason for that must not always be malicious. To assure that the service is not stopped for the wrong reason, the transaction has to be tested for validity again. If it is still valid, the process can continue, if not, the process needs to be stopped.

Seeing that services are going to stop if the transaction is not accepted into the next valid block, it needs to be considered what kind of devices can be offered while accepting pending transactions. A simple case of renting a lock would for example not be applicable as it is. The customer would be able to take the locked object even if the transaction is not mined while the vendor has no control over the rented object. More damaging for the vendor would be the case where a renting request from someone else would be accepted instead. Both customers would have a right for the rented object. Safeguards as discussed in 3.1 need to be put in place to make it reasonably safe for the vendor. Services on the other hand like electricity can be sold without those safeguards. Taking a simple charging station, the electricity would simply stop if the transaction failed. No other customer would be able to practically rent the charging station, because only one car can be connected to the socket at the same time. The financial loss for the vendor is small and the customer can be blocked until the losses are paid for.

The following requirements are defined for accepting pending transactions:

1. Double spending protection is in place either through a deposit or with a price that is acceptable for the vendor to risk
2. The product of the vendor is applicable for the acceptance of pending transactions
3. The transaction is autonomous or there are additional steps in place to verify non-autonomous transactions

4.2 Application

After accepting those criteria, a application is created that applies all findings from the previous chapters. This is done by analyzing needed third party software and its influence on those criteria. The solution is represented through a flow chart diagram.

4.2.1 Ethereum Clients

An Ethereum Clients role is to connect to the Ethereum network by opening up a node. In order to participate in the network, the client needs to implement all rules of the protocol defined in the yellow paper [22]. The protocol is enforced by a set of software tests, which run to validate the correctness of the client. Nevertheless, the actual implementation details are different from client to client.

At the time of writing there are two major implementations of the Ethereum Client. Geth [08], which is developed by the Ethereum foundation provides a Go language implementation. Parity [18] on the other hand is developed by Parity Technologies and is written in Rust. Most commonly used is the Geth client. It is used on around 76% of the nodes. With around 16% follows Parity [12].

Both clients are open source and free to use.

4.2.2 Implemented Transaction Pools

Because the application relies so much on the handling of transactions through the client, the implementation needs to be examined further.

Both, Geth and Parity, implement the transaction pool in a similar fashion. The following is a simplified description of the Geth transaction pool.

The Geth transaction pool consists of a couple of lists which sort the transactions into different categories. Two lists are hold to separate transactions that are executable and transactions which are not. Executable transactions are stored in the pending list while non-executable transactions are stored in the future list. A transaction is executable when all previous transactions are in the list of pending transactions as well or the nonce is indicating that it is in fact the transaction which is to execute next. All transactions that do not fulfill this requirement are stored in the future queue. Geth has a few additional lists for lookup purposes, which do not hold however any transaction that are not in pending or future list. They are separated into a list for all transactions known by the node, a list of local transactions and a list of transactions sorted by price. Figure 4.1 shows what lists are referenced while adding a new pending transaction to the pool.

As soon as a transaction is received by the network, it will be validated against the Ethereum protocol and additional tests specific for Geth. Part of these tests check if

- the transaction is signed properly
- the transaction is not exceeding the block gas limit
- the transaction does not have a lower nonce then the current state nonce of the sender

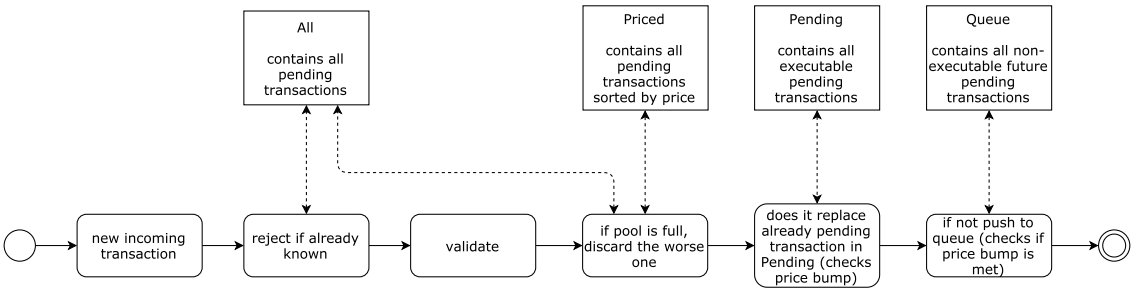

Figure 4.1: Reference on used lists in the transaction pool

- the sender of the transaction has enough funds

After these checks, the transaction is attempted to be put into the future list of transactions. If the transaction is known by the node, it will be dropped. When the future list is already full, then the most unprofitable transaction will be discarded. In case the transaction is replacing a transaction (having the same nonce), then the transaction will be checked against a price bump. The price bump is set by the node owner and describes how high the price difference must be between an already existing transaction and a new incoming transaction. Is the price bump met, the transaction will replace the old one.

An additional function is called to promote transactions from the future list into the pending list. All transactions that became invalid (transactions that have a too low nonce, or transactions that became to costly, meaning that the balance of the transactor became too low or the gas limit became unacceptable) will be discarded and transactions that became executable will be added into the pending list. The mining strategy to decide which transaction is taken into the block at what point is partly decided by these lists. Geth implements it in a way that it groups all transactions from the pending list according to there sender address into heaps. Within the individual heaps, the transactions are sorted by nonce. Now the Block is put together by iterating over the heaps and taking the transaction on top with the highest gas price. The need for such a simple algorithm comes from the necessity to save time, in order to mine faster.

Parity has a similar list system that only differentiates through the order in which it processes the incoming transactions. One big difference between Geth and Parity is that Geth has an implemented miner, whereas Parity does not. Because of that the list of pending transactions has a different pattern. Where Geth groups the transactions to be mined, Parity just lists them in a chronological order. The miner on a Parity node has to sort the transactions himself.

4.2.3 Web3

In order to access the functionalities of the Ethereum Clients, they open up a RPC interface through which important calls can be executed. The Web3 library [09] is a way for JavaScript applications to access these calls. It can connect to the client in different ways, such as http requests, through a web socket or through an IPC file. As soon as a connection is established, the user can now access RPC functionality through the web3 object. This does not only allow an easy access to core Ethereum Client features, but makes it easier to handle smart contracts through abstractions.

In this application, the subscribe function is used to listen to every incoming pending transaction. The function call *web3.eth.subscribe('pending')* opens up an event handler which fires when a transaction becomes pending. As described above, the actual transaction pool is separated into different lists. The event handler is only activated, when a transaction moves into the pending list, either from the future list or directly. That implies that listening to this list only returns transactions that are executable in your current node. Taken into account that the local node does not always receives all the pending transactions that are in the network, a transaction might be in the future list locally, but actually pending in the node of a miner. Filtering those transactions that are in the future list locally but are pending in miner nodes is impossible, given the fact that it is not known from where the next block will come.

Other functionalities of the Web3 object include abstract objects for smart contracts, utility functionality to create and send transactions and catching events from the blockchain.

4.2.4 Structure

The discussed methods are implemented in a Node.js module. It requires at least two parameters for execution. These parameters describe autonomous and non-autonomous transactions. In order to call this function, the user needs to know what transactions in his system are autonomous and which are non-autonomous. Autonomous transactions are passed on by a simple array with a unique signature of the contract function that is called[19]. Non-autonomous transactions are passed in by an object. Figure 4.2 resembles an example build for such an object. Each non-autonomous transaction is identified by its signature. Each signature contains a list of transactions it depends on including the sender and receiver address.

Based on these parameters, every incoming pending transaction will be checked. If the transaction is an autonomous transaction, the function *estimateGas*[20] will be called to

[19] This signature consists of the first 32 bit of the functions hash value. Each function call contains this signature in the beginning of the transactions data part.

[20] This function estimates how much gas this call will need to execute. In a case of an error, all gas will be used up. This means that when estimateGas returns the same value as the gasLimit of the transaction,

```
var nonautonTxArr = {
  '0x60fe47b1': [
    {
      to: '0xF62965281564747ac877624A0F6A7362CFeBCD61',
      data: '0xa6f9dae1',
      from: '0x1554bc5bc64c9c304935ae77aa8cdd3e2ac13ae2'
    },
    {
      to: '0xF62965281564747ac877624A0F6A7362CFeBCD61',
      data: '0xa6f9dae1',
      from: '0x1554bc5bc64c9c304935ae77aa8cdd3e2ac13ae2'
    }
  ],
  '0x71af58c2': [
    {
      to: '0xF62965281564747ac877624A0F6A7362CFeBCD61',
      data: '0xb7a0ebf2',
      from: '0x1554bc5bc64c9c304935ae77aa8cdd3e2ac13ae2'
    },
    {
      to: '0xF62965281564747ac877624A0F6A7362CFeBCD61',
      data: '0xc8b1fc03',
      from: '0x1554bc5bc64c9c304935ae77aa8cdd3e2ac13ae2'
    }
  ]
}
```

Figure 4.2: Example structure of an non-autonomous transaction object containing two transactions.

discern if the transaction is executable against the current block.

In case that the transaction is non-autonomous, the pending list from the transaction pool will be checked against the parameter list. Only if every transaction from that list is in the pending pool or previous blocks, the pending transaction can be accepted. The accepted transactions are temporarily stored. As soon as the next block is received, all accepted pending transactions need to be in that block. If that is not the case, but the transaction is still valid as defined, then the service continues. If it is not included and the transaction is not valid anymore, or the transaction in the block failed, the provided service will be stopped. Figure 4.3 shows a complete flow chart diagram of this process.

One problem in this process is the fact that it cannot separate between required transactions that are in the pending pool and transactions that are required but already mined. The issue lies in the *estimateGas*[21] function. This function can only be called against the current state and not against a pending state. Either all required transactions are in the previous blocks which will let *estimateGas* finish successfully or all are in the pending list which will let the implemented algorithm finish successfully.

it is probably going to fail.

[21] *estimateGas* runs the passed transaction against the current state without changing it. In that way it can say how much gas will be used by the function. Ethereum transactions consume all gas when they throw an exception. That allows the forecast that if all gas is used by the transaction, then there was probably an error.

A different approach is possible through an implemented method provided by the Parity client. Parity allows the user to trace transactions and to analyze the outcome accordingly. This allows the execution of multiple transactions against the current state. If the previous transaction would fail, the vendor knows that he should not accept the pending transaction in question. Implementing this method requires the separation between two sub groups of non-autonomous transactions.

- The first subgroup requires transactions that are only from one sender address. The vendor has to check the nonces of those transactions to ensure that there is no gap in between. Only double spending attacks are applicable in order to intercept the acceptance process.
- The second subgroup are transactions that require transactions from other accounts. Because the vendor has no influence on the order in which the transactions will be mined, the chance that the prediction fails is a lot higher than in the first subgroup.

With the implementation of Parity's trace system, the aforementioned problem with the location of the required transactions can be solved by calling the trace functionality with all required transactions that are already in the pending list. If the trace call is successful, all other required transactions are in the previous block. When it fails, required transactions are missing. An updated flow chart can be seen in Figure 4.4.

This implementation is in that way superior to the previous one, that it does not always has to guess if a transaction will go through. Subgroup one is certain to execute successfully, given that no double spend attacks are expected and subgroup two can be determined in more detail.

4.3 Implementation

The previously created flow chart is now implemented into a nodejs module. Both versions, one for Geth and the other for Parity are explained. Demarcation criteria are set.

4.3.1 Demarcation Criteria / Falsifiability

The provided implementation of the given solution is a proof of concept that does not intend to solve all problems spoken of before. The intend is to provide an application that can within reasonable borders accept unconfirmed transactions. It is only functional if the contracts that are watched over are known to the user. For that reason is the proof of concept based on the smart contract shown in figure 3.4. Only the Geth and the Parity client are supported and need to run with an open IPC (inter process communication) endpoint on the system. The Web3 module is at the time of writing still in beta phase.

For that reason is the introduced implementation for the Parity client not functioning as expected[22]. Further bug fixes of the Web3 module are needed to run that code successfully.

4.3.2 Procedure

The concept is implemented as a Node.js module written in JavaScript. It connects to the Ethereum Client via web3js. Through the web3 functionality of *subscribe*, the web3 object listens to every transaction that enters the pending list of the underlying client. It then checks it against the address of the account that is supposed to be watched. Following the concept shown in the flowchart (Figure 4.3), it checks if the transaction is valid by deciding between autonomous and non-autonomous transaction. For autonomous transactions it runs *estimateGas* call to find out if it will execute. If all the gas is used, then it throws an error. Non-autonomous transactions will trigger a sub function which retrieves the pending list from the client and checks if all required transactions are received by the node.

With the Geth implementation (Figure 4.3), it iterates over all transactions in the pool and compares them with the list of required transactions. In case all required transactions are in the pool, it will return true. If not, then *estimateGas* will be called to determine if the transaction could be executable anyway. As shown before, this does not bring a reliable result as shown in section 4.2.4.

Parity is providing the solution with a RPC call named *trace_callMany* which accepts an array of transactions to execute against the current state. It returns a trace which holds informations about the changes done in the state, gas used and further metadata [19]. By executing all dependent transactions in the right order and checking the trace, it can be predicted to a certain degree if the transaction will go through. Further this allows to trace only a subset of required transactions and saves computational time down the line when it comes to checking previous blocks for required transactions. The aforementioned task of checking old blocks would be difficult to implement, because it cannot be determined which transactions are from an older transaction request and which are the ones actually looked for. With *trace_callMany* in place, this is not necessary at all. All available required transactions can be run against the state, where the last one is the actual transaction that needs to be confirmed. If the trace shows that this transaction will execute successfully then all required transactions that are not in the pending list are already mined.

To achieve that, the implementation will iterate through the pending list and store every required transaction that is available. This array will be passed to the trace function

[22] The subscribe functionality for pending transactions is not working for the Parity Client version 1.7.0 and web3 version 1.0.

which will then determine if the transaction will execute or not (Figure 4.4).

In case it is decided that the transaction is ready to be accepted, it will be stored in a separate array. This is necessary to determine if the transaction was in the end actually taken into the next block. Transactions that are not deemed to be acceptable will be stored as well to check them again as soon as another transaction has come in. They are stored in the potential list. In case a new block arrives, the list of accepted transactions will be checked against this new block. This is done by getting the transaction by its hash from the blockchain and using a side effect of pending transactions. Each pending transaction is missing the blocknumber parameter. The web3 function *getTransaction* is looking through the blockchain and the transaction pool to find the transaction with the provided hash. If a transaction in the accepted list is without a blocknumber, it is not in the new Block and needs to be checked again for validity. New blocks might contain transactions that will make some pending transactions invalid, like through the requirements set in the yellow paper. Those will be filtered by the client and removed from the transaction pool. The same has to be done for the arrays with accepted and potential transactions. To do that, the transaction is run through the above process again where *estimateGas* and *trace_callMany* will filter invalid transactions. Those will be removed and a signal is sent to stop systems that have been triggered through an accepted pending transaction.

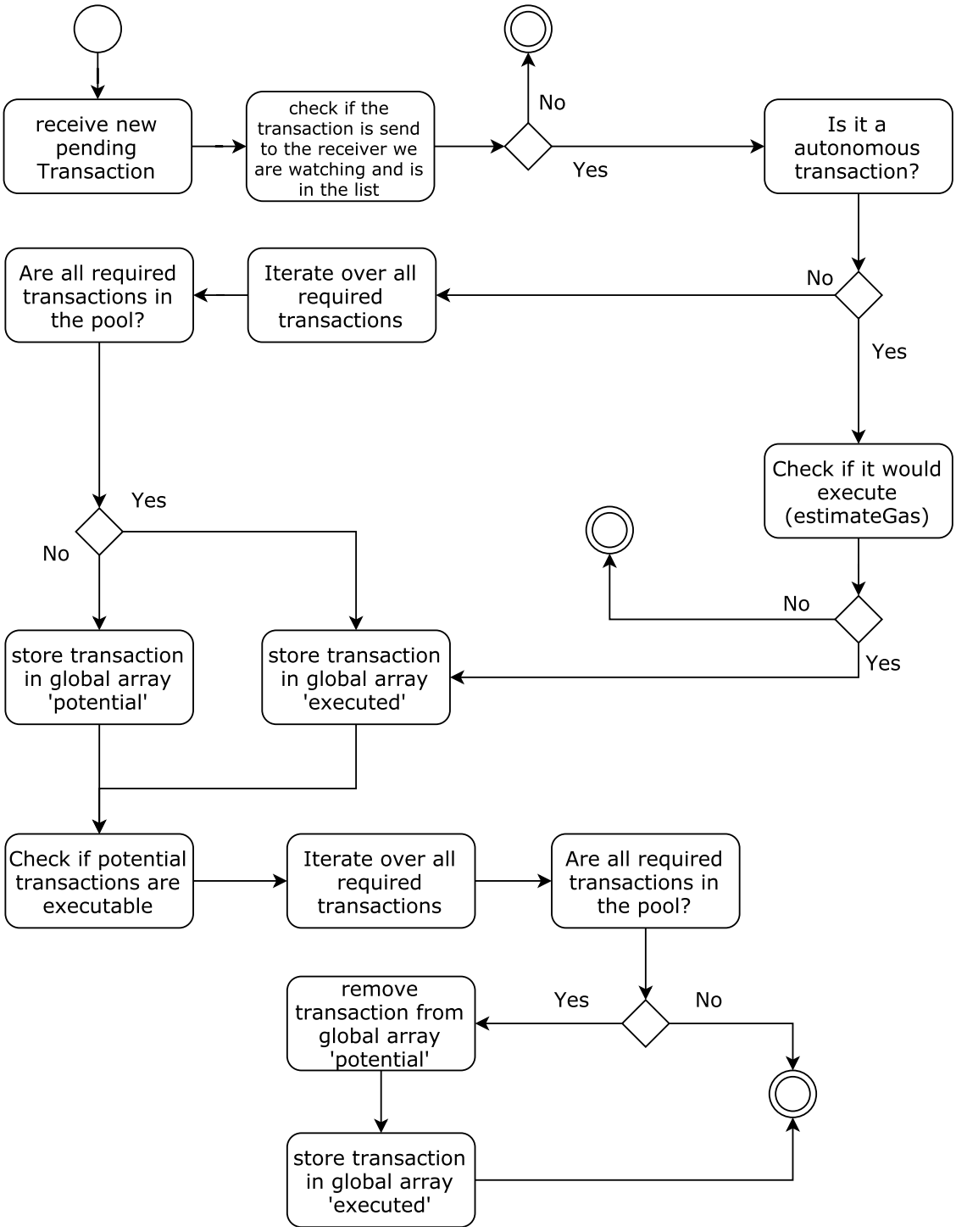

Figure 4.3: Flow chart of the acceptance for pending transactions

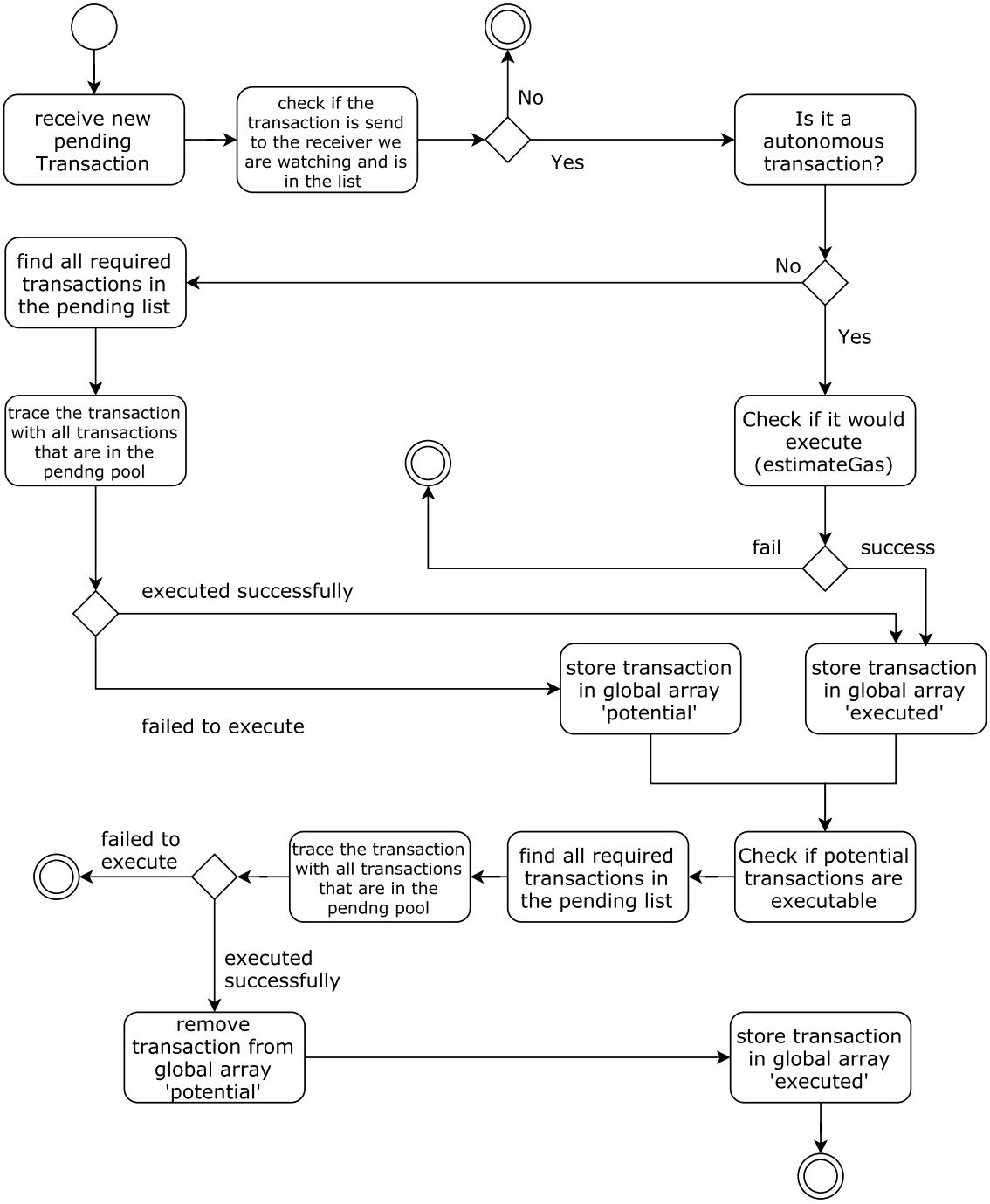

Figure 4.4: Flow chart of the acceptance for pending transactions with Parity's trace functionality

5 Conclusion

This work shows that the question if a vendor should accept pending transactions can not simply be answered with yes or no. Most issues arise through the possibility of double spending attacks and issues with the network topology. While none of those problems is solved in this paper, some workarounds where found that can be used under certain risks and conditions.

Accepting pending transactions requires a detailed analysis of the individual use case. Especially expected customer behavior in form of convenience needs to be reviewed to determine the risk levels of pending transactions. There are groups of use cases that allow a certain degree of freedom in accepting pending transactions. This group has a low transaction value and a high inconvenience level for the customer. Another group allows nearly no freedom at all in accepting pending transactions. That group has a high transaction value and a low inconvenience level. While some use cases might be obvious, some need special attention, especially those who have high inconvenience levels paired with high value and low inconvenience levels paired with low value.

Double spending attacks are even more effective in Ethereum then in Bitcoin and are hard to face. Most clients implement an algorithm that overwrites old transactions, as long as the new transaction has a higher gas price. The introduced solutions from Karame et al for the Bitcoin network only works partially on the Ethereum network and are hard to implement. While those might not apply, smart contracts are able to provide solutions like deposits to keep the vendor save.

To counter the network topology issues, transactions are separated into two groups, autonomous and non-autonomous transactions. While autonomous transactions represent any call that does not rely on other transactions, they can be verified very simply. Non-autonomous transactions need a special algorithm to be accepted. To ensure that all previous transactions are present, the transaction pool needs to be filtered with a given list of transactions. The parity client provides an additional functionality to make checking older blocks easier.

While it can be said that it is reasonably safe to accept pending transactions under certain conditions, it is mostly linked to an unreasonably high risk. The fact that the double spending problem is not sufficiently solved for Ethereum at the moment is making it uneconomical for most use cases. Double spending attacks are even for low value transactions cheap to execute. In the case that this risk is made acceptable with deposits for instance, the created solution enables to verify non-autonomous transactions and accept them in a pending state.

Implementations as shown in this thesis are only a short term solution and by far not

enough for a real world implementation. Methods like parity's textittrace_callMany are crucial for developers to enable save adoption of complex business processes. Future improvements of the Ethereum protocol should implement ways to ensure that the outcome of pending transactions which rely on other pending transactions can be predicted safely.

Bibliography

[01] Alter J.; "When photographs lie", Newsweek, pp. 44-45, July 30, 1990

[02] Arnold A.; "Some Central Banks Are Exploring the Use of Cryptocurrencies", https://www.bloomberg.com/news/articles/2017-06-28/rise-of-digital-coins-has-central-banks-considering-e-versions, 28.06.2017

[03] Bamert T., Decker C., Elsen L., Wattenhofer R., Welten S.; "Have a snack, pay with bitcoins", Peer-to-Peer Computing (P2P),Conference on IEEE, 2013

[04] Bayer, Dave; Haber, Stuart; Stornetta, Scott W.; "Improving the Efficiency and Reliability of Digital Time-Stamping", In: Capocelli R., De Saints A., Vaccaro U. (eds) Sequences II, pp. 329-334, 1992

[05] Buterin V.; "A Next-Generation Smart Contract and Decentralized Application Platform", https://github.com/ethereum/wiki/wiki/White-Paper

[06] Coinmarketcap, "CryptoCurrency Market Capitalizations", https://coinmarketcap.com/, 31.08.2017

[07] Dwork C., Naor M.; "Pricing via processing or combatting junk mail", In 12th Annual International Cryptology Conference, pp. 139-147, 1992

[08] Ethereum Foundation, "Go Ethereum", https://github.com/ethereum/go-ethereum, 31.08.2017

[09] Ethereum Foundation, "web3", https://web3js.readthedocs.io/en/1.0/web3.html, 31.08.2017

[10] Ethereum Foundation, "The Homestead Release", http://www.ethdocs.org/en/latest/introduction/the-homestead-release.html, 31.08.2017

[11] Ethereum Foundation, "Merkle Patricia Trie Specification (also Merkle Patricia Tree)", https://github.com/ethereum/wiki/wiki/Patricia-Tree, 31.08.2017

[12] ethernodes.org, "ethernodes.org", https://www.ethernodes.org/network/1, 07.08.2017

[13] Haber S., Stornetta W. S.; "How to time-stamp a digital document", Journal of Cryptography, Vol. 3, No. 2, pp. 99-111, 1991

[14] Karame G., Androulaki E., Capkun S.; "Two bitcoins at the price of one? double-spending attacks on fast payments in bitcoin", IACR Cryptology ePrint Archive, 2012

[15] Kriss P.; "The Value of Customer Experience, Quantified", https://hbr.org/2014/08/the-value-of-customer-experience-quantified, August 1st 2014, July 18th 2017

[16] Lei M.; "Exploiting Bitcoin's Topology for Double-spend Attacks",

[17] Nakamoto S.; "Bitcoin: A Peer-to-Peer Electronic Cash System", bitcoin.org, Oktober 2008, Retrieved 10. July 2017

[18] Parity Technologies, "Parity", https://github.com/paritytech/parity, 31.08.2017

[19] Parity Technologies, "Multi-call RPC #6195", https://github.com/paritytech/parity/pull/6195, 31.08.2017

[20] Podolanko John P., Ming J., Wright M.; "Countering Double-Spend Attacks on Bitcoin Fast-Pay Transactions", IEEE Symposium on Secrurity and Privacy, 2017

[21] Rosenfeld M.; "Analysis of hashrate-based double-spending", arXiv preprint arXiv:1402.2009, 2012

[22] Wood G.; "Ethereum: A secure decentraliesed generelised transaction ledger EIP-150 Revision" http://gavwood.com/paper.pdf, 13.06.2017

Appendix A: Example Implementation

A.1 Geth Implementation

Following is an example implementation of a Node.js module using Geth as described in section 4.3.

```
var rpc = require('node-json-rpc')
var Web3 = require('web3')
var net = require('net')
var web3 = new Web3('./geth.ipc', net)

var options = {
  port: 8545,
  host: 'localhost',
  path: '/',
  strict: true
}

var client = new rpc.Client(options)

var globalTxStorePotential = []
var globalTxStoreExecuted = []

//changeOwner 0xa6f9dae1
//set 0x60fe47b1

//address that we are scanning
var addressReceiving = '0xF62965281564747ac877624A0F6A7362CFeBCD61'
//array of transactions that are auton
var autonTxArr = ['0xa6f9dae1']
//array of transactions that are non-auton
var nonautonTxArr = {
  '0x60fe47b1': [
    {
      to: '0xF62965281564747ac877624A0F6A7362CFeBCD61',
      data: '0xa6f9dae1',
      from: '0x1554bc5bc64c9c304935ae77aa8cdd3e2ac13ae2'
    },
    {
      to: '0xF62965281564747ac877624A0F6A7362CFeBCD61',
      data: '0xa6f9dae1',
      from: '0x1554bc5bc64c9c304935ae77aa8cdd3e2ac13ae2'
    }
  ],
  '0x71af58c2': [
    {
      to: '0xF62965281564747ac877624A0F6A7362CFeBCD61',
      data: '0xb7a0ebf2',
      from: '0x1554bc5bc64c9c304935ae77aa8cdd3e2ac13ae2'
    },
    {
      to: '0xF62965281564747ac877624A0F6A7362CFeBCD61',
      data: '0xc8b1fc03',
      from: '0x1554bc5bc64c9c304935ae77aa8cdd3e2ac13ae2'
    }
  ]
}
/**
 * Watches the Geth transaction pool for incoming transactions and filters them by given
 * parameters. It also watches new Blocks and acts accordingly.
 *
 * Takes three parameters:
 * _addressReceiving - The address to which the watched transactions are addressed to
 * _autonTxArr       - An array with the first 8 bits of the data from autonomous
 *                     transactions data part
 * _nonautonTxArr    - An array of non-autonomous transactions objects. Those objects are
 *                     build like: {<8bitDataPart>:[<ArrayOfDependentTransactions>]}.
 *
 * Example use for nonautonTxArr:
 *
 * {'0x60fe47b1': [{
 *  to: '0xF62965281564747ac877624A0F6A7362CFeBCD61',
 *  data: '0xa6f9dae1',
 *  from: '0x1554bc5bc64c9c304935ae77aa8cdd3e2ac13ae2'
 *  }]
 * }
 *
 * @param {String} _addressReceiving
```

```
 * @param {Array} _autonTxArr
 * @param {Array} _nonautonTxArr
 */
exports.activateChecker = function(_addressReceiving, _autonTxArr, _nonautonTxArr) {

  addressReceiving = _addressReceiving
  autonTxArr = _autonTxArr
  nonautonTxArr = _nonautonTxArr

  checkNewBlock()

  var subscription = web3.eth.subscribe('pendingTransactions', function(error, result) {
    if (!error) {
      //do something
    }
  })
    .on("data", async function(transaction) {

      //get the whole object when a tx enters the pool
      let tx = await web3.eth.getTransaction(transaction)

      //tx addressed to
      let txto = transaction.to
      //address of the contract we listening to
      let variableTx = addressReceiving
      //parse datatype to be comparable
      let varTxNumber = parseInt(variableTx)
      let txtoNum = parseInt(txto)
      //is this tx addressed to me (the one we listen to)
      if (txtoNum != varTxNumber) {
        //check if the transaction is acceptable
        let booly = await transactionCheck(tx, addressReceiving, autonTxArr, nonautonTxArr)
        if (booly) {
          //save tx to check later when a new block arrives
          globalTxStoreExecuted.push(tx)
          //activate weapon x here
          console.log("Pending Transaction accepted (fire)")
        } else {
          //save tx to check later when new pendings are in
          //store globaly (if a new block arrives it will be emptied by diffrent function)
          globalTxStorePotential.push(tx)
          console.log("Pending Transaction rejected - saved for later")
        }
        //check old saved transactions if their requirements are fullfilled
        console.log("checking if stored transactions are executable")
        for (a of globalTxStorePotential) {
          booly = await transactionCheck(a, addressReceiving, autonTxArr, nonautonTxArr)
          if (booly) {
            //remove from global store
            let index = globalTxStorePotential.indexOf(a)
            if (index > -1)
              globalTxStorePotential.splice(index, 1)
            //save tx to check later when a new block arrives
            globalTxStoreExecuted.push(tx)
            //activate weapon x here
            console.log("Old transaction executed (fire)")
          } else {
            console.log("no old transaction is executable")
          }
        }
      }
    });
}

/**
 * Checks if a transaction is valid/executable. Calls the Client specific RPC call to
 * receive the transaction pool. Takes the transaction that needs to be checked, the
 * receiving Address, and arrays to check against
 *
 * @param {Object} transaction
 * @param {String} addressReceiving
 * @param {Array} autonTxArr
 * @param {Array} nonautonTxArr
 */
async function transactionCheck(transaction, addressReceiving, autonTxArr, nonautonTxArr) {

  //autonomous transactions
  if (autonTxArr.includes(transaction.input.substring(0, 10))) {
    console.log("miniTx")

    let estimatedGasPrice = await web3.eth.estimateGas({
      gas: parseInt(transaction.gas),
      data: transaction.input,
      to: transaction.to,
      from: transaction.from
    })
    if (estimatedGasPrice != transaction.gas)
      return true
```

```
    else
      return false
  }

  //non-autonomous transactions
  let nonautoKeys = Object.keys(nonautonTxArr)
  if (nonautoKeys.includes(transaction.input.substring(0, 10))) {
    console.log("nonAutoTx discovered")
    //tx data suggest that it is non-autonomous (can be replaced with identifier of a contract function)
    //alayse the data :O
    //durchlaufe den txpool and finde alle kollegen
    let retBool = await checkTxPoolComplex_Geth(nonautonTxArr[transaction.input.substring(0, 10)])
    if (retBool)
      return true
    else {
      //check if the transaction would execute anyways (needed transactions might be in the last Block)
      //-->does only work if all required transactions are already mined
      let estimatedGasPrice = await web3.eth.estimateGas({
        gas: parseInt(transaction.gas),
        data: transaction.input,
        to: transaction.to,
        from: transaction.from
      })
      if (estimatedGasPrice != transaction.gas)
        return true
      else
        return false
    }
  }

  return false
}

/**
 * Iterates over the transaction pool and searches for the transactions given in the parameter
 * @param {Array} transactions
 */
function checkTxPoolComplex_Geth(transactions) {
  return client.call({
    "jsonrpc": "2.0",
    "method": "txpool_content",
    "id": 1
  }, function(err, res) {
    if (err) {
      console.log(err)
    } else {
      let booly = false
      let key = Object.keys(transactions)
      //f r jede transaktion in der list die zu checken is
      for (k of key) {
        //see if the address has pending transactions (if not break)
        if (res.result.pending[transactions[k]['from']]) {
          let subKey = Object.keys(res.result.pending[transactions[k]['from']])
          //f r jede transaction in der pending liste der adresse die wir grade checken
          for (s of subKey) {
            //wenn die data gefunden wurde dann mark true und mach mit den n chsten data weiter
            if (res.result.pending[transactions[k]['from']][s]['data'] == transactions[k]['data']) {
              //mark as found
              /**
               * TODO: actually find out if the found transaction is valid :)
               */

              booly = true
              break
            }
          }
        }
        //wenn die data nicht gefunden wurde, gib false zur ck
        if (!booly)
          return false
        else
          booly = false
      }
      return true
    }
  })
}

/**
 * Listens to new blocks and checks if stored transactions are in there
 * -> if a Transaction either from the potential or the executed list is not in the new Block they
 *    will be removed from the list (executed will be moved to potential) and services started by
 *    executed transactions will be stopped
 * */
async function checkNewBlock() {
  web3.eth.subscribe("newBlockHeaders", function(error, result) {
    if (!error) {
      //do something
```

```
    }
  })
    .on("data", async function(blockHeader) {
      //get transaction by transaction hash in the list
      for (tx of globalTxStoreExecuted) {
        //if it has a blocknumber then remove from list
        let rTx = await web3.eth.getTransaction(tx.hash)
        //if the blocknumber is null its still pending
        if (rTx.blockNumber == null) {
          //->check if it is still valid
          rBool = await transactionCheck(rTx, addressReceiving, autonTxArr, nonautonTxArr)
          //if not, then stop service
          if (!rBool) {
            let index = globalTxStoreExecuted.indexOf(tx)
            if (index > -1)
              globalTxStoreExecuted.splice(index, 1)
            //and add to potential
            globalTxStorePotential.push(rTx)
            console.log("Stop that chicken!!!!")
          }

        } else {
          //the transaction is in the new Block
          //-> remove from all internal lists
          let index = globalTxStoreExecuted.indexOf(tx)
          if (index > -1)
            globalTxStoreExecuted.splice(index, 1)
        }
      }

      //do the same with potential transactions
      for (tx of globalTxStorePotential) {
        //if it has a blocknumber then remove from list
        let rTx = await web3.eth.getTransaction(tx.hash)
        //if the blocknumber is null its still pending
        if (rTx.blockNumber == null) {
          //->check if it is still valid
          rBool = await transactionCheck(rTx, addressReceiving, autonTxArr, nonautonTxArr)
          //if not, then stop service
          if (!rBool) {
            let index = globalTxStorePotential.indexOf(tx)
            if (index > -1)
              globalTxStorePotential.splice(index, 1)
          }
        } else {
          //the transaction is in the new Block
          //->delete from list
          let index = globalTxStorePotential.indexOf(tx)
          if (index > -1)
            globalTxStorePotential.splice(index, 1)
        }
      }
    })
}
```

A.2 Parity Implementation

Following is an example implementation of a Node.js module using Parity as described in section 4.3.

```
var rpc = require('node-json-rpc')
var Web3 = require('web3')
var net = require('net')
var web3 = new Web3("ws://localhost:8546")

var options = {
  port: 8545,
  host: 'localhost',
  path: '/',
  strict: true
}

var client = new rpc.Client(options)

var globalTxStorePotential = []
var globalTxStoreExecuted = []

//changeOwner 0xa6f9dae1
//set 0x60fe47b1

//address that we are scanning
```

```
var addressReceiving = '0x6C3f96a0a698B66111c1d5e1520b042d263BdF5e'
//array of transactions that are auton
var autonTxArr = ['0xa6f9dae1']
//array of transactions that are non-auton
var nonautonTxArr = {
  '0x60fe47b1': [{
    to: '0x6C3f96a0a698B66111c1d5e1520b042d263BdF5e',
    data: '0xa6f9dae1',
    from: '0x6d8904bdefd08e6b829d147e6ad12aceafe83b16'
  }]
}

/**
 * Watches the Parity transaction pool for incoming transactions and filters them by given parameters.
 * It also watches new Blocks and acts accordingly.
 *
 * Takes three parameters:
 * _addressReceiving - The address the watched transactions are addressed to
 * _autonTxArr     - An array with the first 8 bits of the data from autonomous transactions data part
 * _nonautonTxArr  - An array of non-autonomous transactions objects. Those objects are build like:
 *                   {8bitDataPart:[ArrayOfDependentTransactions]}.
 *
 * Example use for nonautonTxArr:
 *
 * {'0x60fe47b1': [{
 *  to: '0xF62965281564747ac877624A0F6A7362CFeBCD61',
 *  data: '0xa6f9dae1',
 *  from: '0x1554bc5bc64c9c304935ae77aa8cdd3e2ac13ae2'
 *  }]
 * }
 *
 * @param {String} _addressReceiving
 * @param {Array} _autonTxArr
 * @param {Array} _nonautonTxArr
 */
exports.activateChecker = function(_addressReceiving, _autonTxArr, _nonautonTxArr) {

  addressReceiving = _addressReceiving
  autonTxArr = _autonTxArr
  nonautonTxArr = _nonautonTxArr

  checkNewBlock()

  var subscription = web3.eth.subscribe('pendingTransactions', function(error, result) {
    if (!error) {
      //do something
    }
  })
    .on("data", async function(transaction) {

      //get the whole object when a tx enters the pool
      let tx = await web3.eth.getTransaction(transaction)

      //tx is addressed to me
      let txto = transaction.to
      //address of the contract we listening to
      let variableTx = addressReceiving
      //parse datatype to be comparable
      let varTxNumber = parseInt(variableTx)
      let txtoNum = parseInt(txto)
      //is this tx addressed to me (the one to listen to)
      if (txtoNum != varTxNumber) {
        //check if the transaction is acceptable
        let booly = await transactionCheck(tx, addressReceiving, autonTxArr, nonautonTxArr)
        if (booly) {
          //save tx to check later when a new block arrives
          globalTxStoreExecuted.push(tx)
          //activate weapon x
          console.log("Pending Transaction accepted (fire)")
        } else {
          //save tx to check later when new pendings are in
          //store globaly (if a new block arrives it will be emptied by diffrent function)
          globalTxStorePotential.push(tx)
          console.log("Pending Transaction rejected - saved for later")
        }
        //check old saved transactions if there requirements are in
        console.log("checking if stored transactions are executable")
        for (a of globalTxStorePotential) {
          booly = await transactionCheck(a, addressReceiving, autonTxArr, nonautonTxArr)
          if (booly) {
            //remove from global store
            let index = globalTxStorePotential.indexOf(a)
            if (index > -1)
              globalTxStorePotential.splice(index, 1)
            //save tx to check later when a new block arrives
            globalTxStoreExecuted.push(tx)
            //activate weapon x
            console.log("Old transaction executed (fire)")
```

```
110           } else {
111             console.log("no old transaction is executable")
112           }
113         }
114       }
115     });
116 }
117
118 /**
119  * Checks if a transaction is valid/executable. Calls the Client specific RPC call to receive the transaction pool.
120  * Takes the transaction that needs to be checked, the receiving Address, and arrays to check against
121  *
122  * @param {Object} transaction
123  * @param {String} addressReceiving
124  * @param {Array} autonTxArr
125  * @param {Array} nonautonTxArr
126  */
127 async function transactionCheck(transaction, addressReceiving, autonTxArr, nonautonTxArr) {
128
129   //autonomous transactions
130   if (autonTxArr.includes(transaction.input.substring(0, 10))) {
131     console.log("miniTx")
132
133     //let nonceCheck = await checkNonce(transaction)
134     let estimatedGasPrice = await web3.eth.estimateGas({
135       gas: parseInt(transaction.gas),
136       data: transaction.input,
137       to: transaction.to,
138       from: transaction.from
139     })
140     if (estimatedGasPrice != transaction.gas)
141       return true
142     else
143       return false
144   }
145
146   //non-autonomous transactions
147   let nonautoKeys = Object.keys(nonautonTxArr)
148   if (nonautoKeys.includes(transaction.input.substring(0, 10))) {
149     console.log("nonAutoTx discovered")
150     //tx data suggest that it is non-autonomous (can be replaced with identifier of a contract function)
151     //alayse the data :O
152     //durchlaufe den txpool and finde alle kollegen
153     let foundTx = await checkTxPoolComplex_Parity(nonautonTxArr[transaction.input.substring(0, 10)])
154     //trace all found tx plus receivedTx
155
156     client.call({
157       "jsonrpc": "2.0",
158       "method": "trace_callMany",
159       "id": 1,
160       "params": foundTx
161     }, function(err, res) {
162       if (err) {
163         console.log(err)
164       } else {
165
166         /**
167          * trace_callMany is only implemented in the latest master branch (23.08.2017) and not officialy released
168          * no documentation available
169          *
170          * further information under https://github.com/paritytech/parity/pull/6195
171          *
172          * The following check is not tested and basically pseudo
173          */
174
175         //if the trace of the last transaction indicates that it executed succesfully
176         if (res.result[res.result.length].gasUsed != transaction.gasLimit)
177           return true
178         else
179           return false
180       }
181     })
182     return false
183   }
184 }
185
186 /**
187  * Iterates over the transaction pool and searches for the transactions given in the parameter
188  * @param {Array} transactions
189  */
190 function checkTxPoolComplex_Parity(transactions) {
191   return client.call({
192     "jsonrpc": "2.0",
193     "method": "parity_pendingTransactions",
194     "id": 1
195   }, function(err, res) {
196     if (err) {
197       console.log(err)
```

```
    } else {
      let booly = false
      let foundTx = []

      //f r jede transaktion in der pending liste
      for (tx of res.result) {
        for (pTx of transactions) {
          //compare to each transaction in the transactions list
          if ((tx['input'].substring(0, 10) === pTx['data'].substring(0, 10)) && (tx['from'] === pTx['from'])) {
            //merke die gefundenen
            foundTx.push(pTx)
            //entferne aus der aktuellen liste um schleifenzeit zu verringern
            let index = transactions.indexOf(pTx)
            if (index > -1)
              transactions.splice(index, 1)
          }
        }
      }
      //return array with found transactions
      return foundTx
    }
  })
}

/**
 * Listens to new blocks and checks if stored transactions are in there
 * -> if a Transaction either from the potential or the executed list is not in the new Block they
 *    will be removed from the list (executed will be moved to potential) and services started by
 *    executed transactions will be stopped
 * */
async function checkNewBlock() {
  web3.eth.subscribe("newBlockHeaders", function(error, result) {
    if (!error) {
      //do something
    }
  })
    .on("data", async function(blockHeader) {
      //get transaction by transaction hash in the list
      for (tx of globalTxStoreExecuted) {
        //if it has a blocknumber then remove from list
        let rTx = await web3.eth.getTransaction(tx.hash)
        //if the blocknumber is null its still pending
        if (rTx.blockNumber == null) {
          //->check if it is still valid
          rBool = await transactionCheck(rTx, addressReceiving, autonTxArr, nonautonTxArr)
          //if not, then stop service
          if (!rBool) {
            let index = globalTxStoreExecuted.indexOf(tx)
            if (index > -1)
              globalTxStoreExecuted.splice(index, 1)
            //and add to potential
            globalTxStorePotential.push(rTx)
            console.log("Stop that chicken!!!!")
          }

        } else {
          //the transaction is in the new Block
          //-> remove from all internal lists
          let index = globalTxStoreExecuted.indexOf(tx)
          if (index > -1)
            globalTxStoreExecuted.splice(index, 1)
        }
      }

      //do the same with potential transactions
      for (tx of globalTxStorePotential) {
        //if it has a blocknumber then remove from list
        let rTx = await web3.eth.getTransaction(tx.hash)
        //if the blocknumber is null its still pending
        if (rTx.blockNumber == null) {
          //->check if it is still valid
          rBool = await transactionCheck(rTx, addressReceiving, autonTxArr, nonautonTxArr)
          //if not, then stop service
          if (!rBool) {
            let index = globalTxStorePotential.indexOf(tx)
            if (index > -1)
              globalTxStorePotential.splice(index, 1)
          }
        } else {
          //the transaction is in the new Block
          //->delete from list
          let index = globalTxStorePotential.indexOf(tx)
          if (index > -1)
            globalTxStorePotential.splice(index, 1)
        }
      }
    })
}
```

Erklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 01.09.2017